

Frontiers in Artificial Intelligence and Applications

HANDBOOK

of satisfiability

SECOND EDITION

Editors:

Armin Biere

Marijn Heule

Hans van Maaren

Toby Walsh

IOS
Press

Propositional logic has been recognized throughout the centuries as one of the cornerstones of reasoning in philosophy and mathematics. Over time, its formalization into Boolean algebra was accompanied by the recognition that a wide range of combinatorial problems can be expressed as propositional satisfiability (SAT) problems. Because of this dual role, SAT developed into a mature, multi-faceted scientific discipline, and from the earliest days of computing a search was underway to discover how to solve SAT problems in an automated fashion.

This book, the Handbook of Satisfiability, is the second, updated and revised edition of the book first published in 2009 under the same name. The handbook aims to capture the full breadth and depth of SAT and to bring together significant progress and advances in automated solving. Topics covered span practical and theoretical research on SAT and its applications and include search algorithms, heuristics, analysis of algorithms, hard instances, randomized formulae, problem encodings, industrial applications, solvers, simplifiers, tools, case studies and empirical results. SAT is interpreted in a broad sense, so as well as propositional satisfiability, there are chapters covering the domain of quantified Boolean formulae (QBF), constraints programming techniques (CSP) for word-level problems and their propositional encoding, and satisfiability modulo theories (SMT). An extensive bibliography completes each chapter.

This second edition of the handbook will be of interest to researchers, graduate students, final-year undergraduates, and practitioners using or contributing to SAT, and will provide both an inspiration and a rich resource for their work.

Edmund Clarke, 2007 ACM Turing Award Recipient

“SAT solving is a key technology for 21st century computer science.”

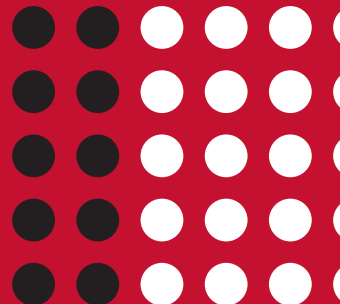
Donald Knuth, 1974 ACM Turing Award Recipient

“SAT is evidently a killer app, because it is key to the solution of so many other problems.”

Stephen Cook, 1982 ACM Turing Award Recipient

“The SAT problem is at the core of arguably the most fundamental question in computer science: What makes a problem hard?”

ISBN 978-1-64368-160-3 (print)
ISBN 978-1-64368-161-0 (online)
ISSN 0922-6389 (print)
ISSN 1879-8314 (online)



HANDBOOK OF SATISFIABILITY

Frontiers in Artificial Intelligence and Applications

The book series Frontiers in Artificial Intelligence and Applications (FAIA) covers all aspects of theoretical and applied Artificial Intelligence research in the form of monographs, selected doctoral dissertations, handbooks and proceedings volumes. The FAIA series contains several sub-series, including ‘Information Modelling and Knowledge Bases’ and ‘Knowledge-Based Intelligent Engineering Systems’. It also includes the biennial European Conference on Artificial Intelligence (ECAI) proceedings volumes, and other EurAI (European Association for Artificial Intelligence, formerly ECCAI) sponsored publications. The series has become a highly visible platform for the publication and dissemination of original research in this field. Volumes are selected for inclusion by an international editorial board of well-known scholars in the field of AI. All contributions to the volumes in the series have been peer reviewed.

The FAIA series is indexed in ACM Digital Library; DBLP; EI Compendex; Google Scholar; Scopus; Web of Science: Conference Proceedings Citation Index – Science (CPCI-S) and Book Citation Index – Science (BKCI-S); Zentralblatt MATH.

Series Editors:

Joost Breuker, Nicola Guarino, Pascal Hitzler, Joost N. Kok, Jiming Liu,
Ramon López de Mántaras, Riichiro Mizoguchi, Mark Musen, Sankar K. Pal,
Ning Zhong

Volume 336

Recently published in this series

- Vol. 335. M. Nørskov, J. Seibt and O.S. Quick (Eds.), Culturally Sustainable Social Robotics – Proceedings of Robophilosophy 2020 – August 18–21, 2020, Aarhus University and online
- Vol. 334. S. Villata, J. Harašta and P. Křemen (Eds.), Legal Knowledge and Information Systems – JURIX 2020: The Thirty-third Annual Conference, Brno, Czech Republic, December 9–11, 2020
- Vol. 333. M. Tropmann-Frick, B. Thalheim, H. Jaakkola, Y. Kiyokir and N. Yoshida (Eds.), Information Modelling and Knowledge Bases XXXII
- Vol. 332. A.J. Tallón-Ballesteros and C.-H. Chen (Eds.), Machine Learning and Artificial Intelligence – Proceedings of MLIS 2020
- Vol. 331. A.J. Tallón-Ballesteros (Ed.), Fuzzy Systems and Data Mining VI – Proceedings of FSDM 2020
- Vol. 330. B. Brodaric and F. Neuhaus (Eds.), Formal Ontology in Information Systems – Proceedings of the 11th International Conference (FOIS 2020)
- Vol. 329. A.J. Tallón-Ballesteros (Eds.), Modern Management based on Big Data I – Proceedings of MMBD 2020

ISSN 0922-6389 (print)

ISSN 1879-8314 (online)

Handbook of Satisfiability

Second Edition

Edited by

Armin Biere

Johannes Kepler University, Linz, Austria

Marijn Heule

Carnegie Mellon University, Pittsburgh, USA

Hans van Maaren

TU Delft, The Netherlands, Delft

and

Toby Walsh

University of New South Wales, Kensington, Australia

Part 1/Part 2

IOS
Press

Amsterdam • Berlin • Washington, DC

© 2021 The authors and IOS Press.

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, without prior written permission from the publisher.

ISBN 978-1-64368-160-3 (print)

ISBN 978-1-64368-161-0 (online)

Library of Congress Control Number: 2021931050

doi: 10.3233/FAIA336

Publisher

IOS Press BV

Nieuwe Hemweg 6B

1013 BG Amsterdam

Netherlands

fax: +31 20 687 0019

e-mail: order@iospress.nl

For book sales in the USA and Canada:

IOS Press, Inc.

6751 Tepper Drive

Clifton, VA 20124

USA

Tel.: +1 703 830 6300

Fax: +1 703 830 2300

sales@iospress.com

LEGAL NOTICE

The publisher is not responsible for the use which might be made of the following information.

PRINTED IN SLOVENIA

Preface First Edition

Propositional logic has been recognized throughout the centuries as one of the corner stones of reasoning in philosophy and mathematics. During history, its formalization into Boolean algebra was gradually accompanied by the recognition that a wide range of combinatorial problems can be expressed as propositional satisfiability (SAT) problems. Because of these two roles, SAT has become a mature multi-faceted scientific discipline. Initiated by the work of Cook, who established its \mathcal{NP} -complete status in 1971, SAT has become a reference problem for an enormous variety of complexity statements.

Simultaneously, many real-world problems were formalized as SAT decision problems using different encoding techniques. This includes verification problems in hardware and software, planning, scheduling and combinatorial design.

Due to the potential practical implications, an intensive search from the early days of computing has been underway of how one could solve SAT problems in an automated fashion. In 1957, Allen Newell and Herb Simon introduced one of the first AI programs, the Logic Theory Machine, to prove propositional theorems from Whitehead and Russell's "Principia mathematica". Shortly after in 1960, Martin Davis and Hillary Putnam introduced their now famous decision procedure for propositional satisfiability problems (a more space efficient version due to Martin Davis, George Logemann and Donald Loveland followed in 1962).

Asked by the editors of this handbook as a witness of his role in this development Martin Davis wrote

I felt delighted and honored when I was asked to write a foreword to this Handbook. But when I examined the table of contents and especially, when I read the initial historical article, I concluded that it is an honor that I hardly deserve. When Hilary Putnam and I began our work on automated theorem proving half a century ago, we mainly thought of it as a way to get funding for work on our real passion, Hilbert's tenth problem. We hardly imagined that SAT might have an independent importance aside from its role in automated deduction. It is wonderful to see what a flourishing area of research satisfiability has become.

Of course what is regarded as the most important problem in theoretical computer science, $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ lives right here. It is remarkable that we still don't know whether or not there is a poly-time algorithm for SAT. I am in a tiny minority in being unconvinced that the answer must be "No". I surely don't expect anyone to find a really efficient uniform algorithm for SAT. But I find the heuristic arguments against the existence of a very inefficient algorithm for SAT that happens to run in poly-time quite unconvincing.

The topics of the handbook span practical and theoretical research on SAT and its applications and include search algorithms, heuristics, analysis of algorithms, hard instances, randomized formulae, problem encodings, industrial applications, solvers, simplifiers, tools, case studies and empirical results. SAT is

interpreted in a rather broad sense. Besides propositional satisfiability it includes the domain of quantified Boolean formulae (QBF), constraints programming techniques (CSP) for word-level problems and their propositional encoding and particularly satisfiability modulo theories (SMT).

The handbook aims to capture the full breadth and depth of SAT and to bundle significant progress and advances in automated solving. It covers the main notions and techniques and introduces various formal extensions. Each area is dealt with in a survey-like style, where some details may be neglected in favor of coverage. The extensive bibliography concluding each chapter will help the interested reader to find his way to master necessary details.

The intended audience of the handbook consists of researchers, graduate students, upper-year undergraduates, and practitioners who wish to learn about the state of the art in actual solving. Limited prior knowledge about the field is assumed. The handbook also aims to assist researchers from other fields in applying ideas and methods to their own work. We thus hope the handbook will provide a means for cross fertilization.

The start and completion of the underlying project would not have been possible without the support and encouragement of many people. As such, this handbook is a community effort and we take this opportunity to express our gratitude to this community as a whole, without addressing to all individual contributions. We are grateful to IOS Press Publishing Company because of their interest in materializing all these efforts into the book you are holding now.

We are indebted to Martin Davis for his contribution to this preface and to Edmund Clarke for his support and recommendations which you may find at the back cover of this copy.

Along with the completion of this handbook another similar project was started by Teofilo Gonzalez. The handbook on \mathcal{NP} -Completeness, Theory and Applications, will be finished shortly and could be considered as a very welcome addition to those concepts which touch propositional proof complexity. These typically proof-system related issues are certainly not fully covered by us. And also there, various practical contributions on SAT solving will find updates which continue to be relevant as the discipline moves forward.

Finally we take the opportunity to spend a few words on a particularly inspiring event who took place during the last SAT conferences: The SAT Competition. Started at the Cincinnati conference in 2002, Daniel Le Berre and Laurent Simon put an incredible amount of work in establishing the format of rules, benchmarks and solver-performance evaluations which made SAT a competitive area where especially young researchers feel triggered to contribute. As a consequence, one of the most frequently used references in this handbook undoubtedly is their competition web page.

Arjen van Lith designed the cover of our Handbook. It is a challenge to recognize various SAT research related patterns hidden there.

*Armin Biere
Marijn Heule
Hans van Maaren
Toby Walsh*

Preface Second Edition

When the first edition of this handbook was published in 2009 advances in SAT were mostly only known to experts, and SAT was seen as a key technology only in certain applications. Since then the number of practical applications has exploded, along with greater awareness of the usefulness of SAT in general: In the last century showing that a certain problem is as hard as SAT was the end of the story and trying to solve it directly seemed to be hopeless. Relying on the continuing improvements made in practical SAT solving, it is now widely accepted that being able to encode a problem into SAT also is highly likely to lead to a practical solution. This “SAT Revolution” started at the end of the last century and continues to produce amazing new practical and theoretical results.

Accordingly, this second edition contains several updates, including a completely revamped Chapter 4 on conflict-driven clause learning. Half the chapters were updated or extended or both. Comments by Donald Knuth were also taken into account, collected while he was preparing the Section on Satisfiability in volume 4 of the “The Art of Computer Programming”. This section appeared as fascicle 6a with more than 300 pages in 2015 and is a major milestone in the SAT literature that has appeared since the first edition of this Handbook.

Three important topics, which were already in the first edition of the handbook and deserve their own chapter, are now given enough space and discussed in detail: there is the new Chapter 7 on proof complexity, Chapter 9 on preprocessing, as well as the new Chapter 12 on automated configuration and solver selection. Additionally, the new Chapter 15 covers proofs of unsatisfiability, one of the main recent developments in practical SAT solving. These proofs are essential in solving long-standing mathematical problems.

Beside these four completely new chapters, there are three new chapters which cover topics already discussed in the first edition. These three chapters focus on new aspects and new technology and are written by authors who made fundamental contributions to change the state of the art. First, recent developments regarding quantified Boolean formulas including the discussion of proof systems are covered in the new Chapter 31. Second, the focus of the new Chapter 24 is on core-based methods for maximum satisfiability which have improved scalability considerably. Finally, the last new Chapter 26 covers a novel research direction on practical and approximate model counting with strong statistical guarantees.

Research in SAT has established itself as a vibrant cross community effort. Beside the main SAT conference, other major conferences and journals in diverse fields from automated reasoning, verification, hardware to software engineering, complexity theory, even algorithms and of course artificial intelligence cover SAT

and its extensions areas of interest. It is also worth mentioning that competitions continue to serve as a show case as well as motivation of the field.

One could argue that SAT is now very widespread across these fields. As one example, and maybe most striking, is the adoption of SAT in other reasoning disciplines, starting of course with SMT solving, the use of lazy clause generation in constraint programming, as well as SAT based splitting in first-order theorem proving, and finally in the use of hammers in interactive theorem proving for higher-order logic relying directly or indirectly on SAT.

As the first edition appears to have done, we hope that also this second edition of the Handbook will serve researchers and practitioners using or contributing to SAT, and provide both an inspiration and a rich resource for their own work.

Armin Biere
Marijn Heule
Hans van Maaren
Toby Walsh

Contents

Part I. Theory and Algorithms

Chapter 1. A History of Satisfiability	3
<i>John Franco and John Martin</i>	
1.1 Preface: the concept of satisfiability	3
1.2 The ancients	6
1.3 The medieval period	8
1.4 The renaissance	9
1.5 The first logic machine	10
1.6 Boolean algebra	10
1.7 Frege, logicism, and quantification logic	12
1.8 Russell and Whitehead	13
1.9 Gödel's incompleteness theorem	14
1.10 Effective process and recursive functions	14
1.11 Herbrand's theorem	15
1.12 Model theory and Satisfiability	15
1.13 Completeness of first-order logic	17
1.14 Application of logic to circuits	18
1.15 Resolution	19
1.16 The complexity of resolution	21
<i>Alasdair Urquhart</i>	
1.17 Refinement of Resolution-Based SAT Solvers	23
1.18 Upper bounds	25
<i>Ewald Speckenmeyer</i>	
1.19 Classes of easy expressions	27
1.20 Binary Decision Diagrams	31
1.21 Probabilistic analysis: SAT algorithms	32
1.22 Probabilistic analysis: thresholds	39
1.23 Stochastic Local Search	42
<i>Holger Hoos</i>	
1.24 Maximum Satisfiability	43
<i>Hantao Zhang</i>	
1.25 Nonlinear formulations	45
<i>Miguel Anjos</i>	
1.26 Pseudo-Boolean Forms	49
1.27 Quantified Boolean formulas	51
<i>Hans Kleine Büning</i>	
References	55

Chapter 2. CNF Encodings	75
<i>Steven Prestwich</i>	
2.1 Introduction	75
2.2 Transformation to CNF	75
2.3 Case studies	83
2.4 Desirable properties of CNF encodings	91
2.5 Conclusion	94
References	94
Chapter 3. Complete Algorithms	101
<i>Adnan Darwiche and Knot Pipatsrisawat</i>	
3.1 Introduction	101
3.2 Technical Preliminaries	101
3.3 Satisfiability by Existential Quantification	104
3.4 Satisfiability by Inference Rules	109
3.5 Satisfiability by Search: The DPLL Algorithm	112
3.6 Satisfiability by Combining Search and Inference	116
3.7 Conclusions	128
References	128
Chapter 4. CDCL SAT Solving	133
<i>Joao Marques-Silva, Ines Lynce and Sharad Malik</i>	
4.1 Introduction	133
4.2 Preliminaries	135
4.3 Implementing CDCL SAT Solvers	140
4.4 Using CDCL SAT Solvers	152
4.5 Impact of CDCL SAT Solvers	160
4.6 Historical Perspective	161
4.7 To Probe Further	163
4.8 Conclusions & Research Directions	164
References	164
Chapter 5. Look-Ahead Based SAT Solvers	183
<i>Marijn J.H. Heule and Hans van Maaren</i>	
5.1 Introduction	183
5.2 General and Historical Overview	185
5.3 Heuristics	192
5.4 Additional Reasoning	201
5.5 Eager Data-Structures	207
References	210
Chapter 6. Incomplete Algorithms	213
<i>Henry Kautz, Ashish Sabharwal and Bart Selman</i>	
6.1 Greedy Search and Focused Random Walk	215
6.2 Extensions of the Basic Local Search Method	218
6.3 Discrete Lagrangian Methods	219
6.4 The Phase Transition Phenomenon in Random k -SAT	222
6.5 A New Technique for Random k -SAT: Survey Propagation	224

6.6	Conclusion	225
	References	226
Chapter 7. Proof Complexity and SAT Solving		233
	<i>Sam Buss and Jakob Nordström</i>	
7.1	Introduction	233
7.2	Preliminaries	236
7.3	Resolution and CDCL SAT solvers	238
7.4	Resolution and Proof Complexity	255
7.5	Algebraic Proof Systems	270
7.6	Cutting Planes and Pseudo-Boolean Solving	282
7.7	Cutting Planes and Proof Complexity	300
7.8	Extended Resolution and DRAT Proof Systems	312
7.9	Frege and Extended Frege Proof Systems	316
7.10	Bounded-Depth Frege Proof System	318
7.11	Concluding Remarks	322
	References	324
Chapter 8. Fundamentals of Branching Heuristics		351
	<i>Oliver Kullmann</i>	
8.1	Introduction	351
8.2	A general framework for branching algorithms	353
8.3	Branching tuples and the canonical projection	357
8.4	Estimating tree sizes	362
8.5	Axiomatising the canonical order on branching tuples	368
8.6	Alternative projections for restricted branching width	369
8.7	How to select distances and measures	371
8.8	Optimising distance functions	378
8.9	The order of branches	380
8.10	Beyond clause-sets	383
8.11	Conclusion and outlook	385
	References	386
Chapter 9. Preprocessing in SAT Solving		391
	<i>Armin Biere, Matti Järvisalo and Benjamin Kiesl</i>	
9.1	Introduction	391
9.2	Classical Preprocessing Techniques	394
9.3	Resolution-Based Preprocessing	401
9.4	CNF Preprocessing Beyond Resolution	409
9.5	Solution Reconstruction	411
9.6	Structure-Based Preprocessing	414
9.7	Conclusion	418
	References	419
Chapter 10. Random Satisfiability		437
	<i>Dimitris Achlioptas</i>	
10.1	Introduction	437
10.2	The State of the Art	439
10.3	Random MAX k -SAT	441

10.4	Physical Predictions for Solution-space Geometry	444
10.5	The Role of the Second Moment Method	447
10.6	Generative models	447
10.7	Algorithms	451
10.8	Belief/Survey Propagation and the Algorithmic Barrier	454
10.9	Backtracking Algorithms	455
10.10	Exponential Running-Time for $k > 3$	457
	References	458
Chapter 11. Exploiting Runtime Variation in Complete Solvers		463
	<i>Carla P. Gomes and Ashish Sabharwal</i>	
11.1	Runtime Variation in Backtrack Search	465
11.2	Exploiting Runtime Variation: Randomization and Restarts	472
11.3	Conclusion	477
	References	477
Chapter 12. Automated Configuration and Selection of SAT Solvers		481
	<i>Holger H. Hoos, Frank Hutter and Kevin Leyton-Brown</i>	
12.1	Introduction	481
12.2	Algorithm configuration	482
12.3	Per-instance algorithm selection	489
12.4	Related approaches	496
12.5	Conclusions and open challenges	499
	References	500
Chapter 13. Symmetry and Satisfiability		509
	<i>Karem A. Sakallah</i>	
13.1	Motivating Example	509
13.2	Preliminaries	512
13.3	Group Theory Basics	516
13.4	CNF Symmetry	525
13.5	Automorphism Group of a Colored Graph	526
13.6	Detection of CNF Symmetries	531
13.7	Symmetry Breaking	537
13.8	From Symmetry to Satisfiability and Back	548
13.9	Summary and a Look Forward	557
13.10	Bibliographic Notes	560
	References	566
Chapter 14. Minimal Unsatisfiability and Autarkies		571
	<i>Hans Kleine Büning and Oliver Kullmann</i>	
14.1	Introduction	571
14.2	Deficiency	572
14.3	Resolution and Homomorphism	575
14.4	Special Classes	577
14.5	Extension to non-clausal formulas	580
14.6	Minimal Falsity for QBF	583
14.7	Applications and Experimental Results	585

14.8	Generalising satisfying assignments through “autarkies”	585
14.9	The autarky monoid	591
14.10	Finding and using autarkies	598
14.11	Autarky systems: Using weaker forms of autarkies	605
14.12	Connections to combinatorics	617
14.13	Generalisations and extensions of autarkies	624
14.14	Conclusion	627
	References	627
Chapter 15. Proofs of Unsatisfiability		635
	<i>Marijn J.H. Heule</i>	
15.1	Introduction	635
15.2	Proof Systems	637
15.3	Proof Search	644
15.4	Proof Formats	645
15.5	Proof Production in Practical SAT Solving	653
15.6	Proof Validation and Processing	657
15.7	Proof Applications	660
15.8	Conclusions	661
	References	661
Chapter 16. Worst-Case Upper Bounds		669
	<i>Evgeny Dantsin and Edward A. Hirsch</i>	
16.1	Preliminaries	669
16.2	Tractable and intractable classes	671
16.3	Upper bounds for k -SAT	674
16.4	Upper bounds for General SAT	679
16.5	How large is the exponent?	682
16.6	Summary table	686
16.7	Addendum for the 2nd Edition: Connections to Circuit Complexity	687
	References	688
Chapter 17. Fixed-Parameter Tractability		693
	<i>Marko Samer and Stefan Szeider</i>	
17.1	Introduction	693
17.2	Preliminaries	695
17.3	Parameterized SAT	700
17.4	Backdoor Sets	704
17.5	Treewidth	712
17.6	Further Satisfiability Parameters	721
17.7	Concluding Remarks	726
	References	727
Part II. Applications and Extensions		
Chapter 18. Bounded Model Checking		739
	<i>Armin Biere</i>	
18.1	Model Checking	739

18.2	Bounded Semantics	742
18.3	Propositional Encodings	743
18.4	Completeness	747
18.5	Induction	749
18.6	Interpolation	750
18.7	Completeness with Interpolation	753
18.8	Invariant Strengthening	755
18.9	Related Work	755
18.10	Conclusion	756
	References	757
Chapter 19. Planning and SAT		765
	<i>Jussi Rintanen</i>	
19.1	Introduction	765
19.2	Classical Planning	766
19.3	Sequential Plans	767
19.4	Parallel Plans	770
19.5	Finding a Satisfiable Formula	775
19.6	Temporal Planning	779
19.7	Contingent Planning	781
	References	785
Chapter 20. Software Verification		791
	<i>Daniel Kroening</i>	
20.1	Programs use Bit-Vectors	791
20.2	Formal Models of Software	792
20.3	Turning Bit-Vector Arithmetic into CNF	795
20.4	Bounded Model Checking for Software	798
20.5	Predicate Abstraction using SAT	803
20.6	Conclusion	814
	References	815
Chapter 21. Combinatorial Designs by SAT Solvers		819
	<i>Hantao Zhang</i>	
21.1	Introduction	819
21.2	Combinatorial Design Problems	821
21.3	Encoding Design Theory Problems	845
21.4	Conclusions and Open Problems	852
	References	853
Chapter 22. Connections to Statistical Physics		859
	<i>Fabrizio Altarelli, Rémi Monasson,</i> <i>Guilhem Semerjian and Francesco Zamponi</i>	
22.1	Introduction	859
22.2	Phase Transitions: Basic Concepts and Illustration	861
22.3	Phase transitions in random CSPs	869
22.4	Local search algorithms	878
22.5	Decimation based algorithms	882
22.6	Conclusion	894

References	895
Chapter 23. MaxSAT	903
<i>Chu Min Li and Felip Manyà</i>	
23.1 Introduction	903
23.2 Preliminaries	904
23.3 Branch and Bound Algorithms	906
23.4 Complete Inference in MaxSAT	913
23.5 Approximation Algorithms	917
23.6 The MaxSAT Evaluation	917
23.7 Other Contributions to MaxSAT	918
23.8 The MinSAT Problem	918
23.9 Conclusions	919
References	920
Chapter 24. Maximum Satisfiability	929
<i>Fahiem Bacchus, Matti Järvisalo and Ruben Martins</i>	
24.1 Introduction	929
24.2 The MaxSAT Formalism	931
24.3 Encodings and Applications	933
24.4 Modern MaxSAT Algorithms	939
24.5 Further Developments	966
24.6 Summary	971
References	972
Chapter 25. Model Counting	993
<i>Carla P. Gomes, Ashish Sabharwal and Bart Selman</i>	
25.1 Computational Complexity of Model Counting	994
25.2 Exact Model Counting	996
25.3 Approximate Model Counting	1003
25.4 Conclusion	1010
References	1011
Chapter 26. Approximate Model Counting	1015
<i>Supratik Chakraborty, Kuldeep S. Meel and Moshe Y. Vardi</i>	
26.1 Introduction	1015
26.2 Approximate Model Counting for CNF	1020
26.3 Handling CNF+XOR Constraints	1030
26.4 Approximate Model Counting for DNF	1032
26.5 Weighted Counting	1035
26.6 Conclusion	1039
References	1040
Chapter 27. Non-Clausal SAT and ATPG	1047
<i>Rolf Drechsler, Tommi Junttila and Ilkka Niemelä</i>	
27.1 Introduction	1047
27.2 Basic Definitions	1048
27.3 Satisfiability Checking for Boolean Circuits	1051
27.4 Automatic Test Pattern Generation	1064
27.5 Conclusions	1080

References	1080
Chapter 28. Pseudo-Boolean and Cardinality Constraints	1087
<i>Olivier Roussel and Vasco Manquinho</i>	
28.1 Introduction	1087
28.2 Basic Definitions	1088
28.3 Decision Problem versus Optimization Problem	1091
28.4 Expressive Power of Cardinality and Pseudo-Boolean Constraints	1093
28.5 Inference Rules	1095
28.6 Current Algorithms	1102
28.7 Conclusion	1125
References	1125
Chapter 29. QBF Theory	1131
<i>Hans Kleine Büning and Uwe Bubeck</i>	
29.1 Introduction	1131
29.2 Syntax and Semantics	1131
29.3 Complexity Results	1137
29.4 Models and Expressive power	1140
29.5 Q-Resolution	1146
29.6 Quantified Horn Formulas and Q2-CNF	1150
References	1154
Chapter 30. QBFs reasoning	1157
<i>Enrico Giunchiglia, Paolo Marin and Massimo Narizzano</i>	
30.1 Introduction	1157
30.2 Quantified Boolean Logic	1157
30.3 Applications of QBFs and QBF reasoning	1158
30.4 QBF solvers	1159
30.5 Other approaches, extensions and conclusions	1172
References	1172
Chapter 31. Quantified Boolean Formulas	1177
<i>Olaf Beyersdorff, Mikoláš Janota, Florian Lonsing and Martina Seidl</i>	
31.1 Introduction	1177
31.2 Preliminaries	1178
31.3 Proof Systems Based on Q-Resolution	1179
31.4 Expansion-Based Proof Systems	1191
31.5 Preprocessing	1198
31.6 Extraction of Winning Strategies from Proofs	1204
31.7 Connections Between Proof Systems	1207
References	1215
Chapter 32. SAT Techniques for Modal and Description Logics	1223
<i>Roberto Sebastiani and Armando Tacchella</i>	
32.1 Introduction	1223
32.2 Background	1225
32.3 Basic Modal DPLL	1231
32.4 Advanced Modal DPLL	1242

32.5	The OBDD-based Approach	1252
32.6	The Eager DPLL-based approach	1256
	References	1261
Chapter 33. Satisfiability Modulo Theories		1267
	<i>Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia and Cesare Tinelli</i>	
33.1	Introduction	1267
33.2	Background	1269
33.3	Eager Encodings to SAT	1275
33.4	Integrating Theory Solvers into SAT Engines	1285
33.5	Theory Solvers	1297
33.6	Combining Theories	1303
33.7	Extensions and Enhancements	1310
	References	1316
Chapter 34. Stochastic Boolean Satisfiability		1331
	<i>Stephen M. Majercik</i>	
34.1	Introduction	1331
34.2	Definitions and Notation	1331
34.3	Complexity of SSAT and Related Problems	1334
34.4	Applications	1335
34.5	Analytical Results	1336
34.6	Algorithms and Empirical Results	1337
34.7	Stochastic Constraint Programming	1354
34.8	Future Directions	1356
	References	1363
Subject Index		1371
Cited Author Index		1429
Contributing Authors and Affiliations		1463

This page intentionally left blank

Part I

Theory and Algorithms

This page intentionally left blank

Chapter 1

A History of Satisfiability

John Franco and John Martin

with sections contributed by Miguel Anjos, Holger Hoos, Hans Kleine Büning,
Ewald Speckenmeyer, Alasdair Urquhart, and Hantao Zhang

1.1. Preface: the concept of satisfiability

Interest in Satisfiability is expanding for a variety of reasons, not in the least because nowadays more problems are being solved faster by SAT solvers than other means. This is probably because Satisfiability stands at the crossroads of logic, graph theory, computer science, computer engineering, and operations research. Thus, many problems originating in one of these fields typically have multiple translations to Satisfiability and there exist many mathematical tools available to the SAT solver to assist in solving them with improved performance. Because of the strong links to so many fields, especially logic, the history of Satisfiability can best be understood as it unfolds with respect to its logic roots. Thus, in addition to time-lining events specific to Satisfiability, the chapter follows the presence of Satisfiability in logic as it was developed to model human thought and scientific reasoning through its use in computer design and now as modeling tool for solving a variety of practical problems. In order to succeed in this, we must introduce many ideas that have arisen during numerous attempts to reason with logic and this requires some terminology and perspective that has developed over the past two millennia. It is the purpose of this preface to prepare the reader with this information so as to make the remainder of the chapter more understandable and enlightening.

Logic is about *validity* and *consistency*. The two ideas are interdefinable if we make use of negation (\neg): the argument from p_1, \dots, p_n to q is valid if and only if the set $\{p_1, \dots, p_n, \neg q\}$ is inconsistent. Thus, validity and consistency are really two ways of looking at the same thing and each may be described in terms of syntax or semantics.

The syntactic approach gives rise to *proof theory*. Syntax is restricted to definitions that refer to the syntactic form (that is, grammatical structure) of the sentences in question. In proof theory the term used for the syntactic version of validity is *derivability*. Proofs are derived with respect to an *axiom system* which is defined syntactically as consisting of a set of axioms with a specified

grammatical form and a set of inference rules that sanction proof steps with specified grammatical forms. Given an axiom system, derivability is defined as follows: q is derivable from p_1, \dots, p_n (in symbols, $p_1, \dots, p_n \vdash q$) if and only if there is a proof in the axiom system of q (derivable) from p_1, \dots, p_n . Because the axioms and rules are defined syntactically, so is the notion of derivability. The syntactic version of consistency is simply called *consistency*, and is defined as follows: $\{p_1, \dots, p_n\}$ is *consistent* if and only if it is not possible to derive a contradiction from $\{p_1, \dots, p_n\}$. It follows that $\{p_1, \dots, p_n\}$ is *inconsistent* if and only if there is some contradiction $q \wedge \neg q$ such that $\{p_1, \dots, p_n\} \vdash q \wedge \neg q$. Since derivability has a definition that only makes reference to the syntactic shapes, and since consistency is defined in terms of derivability, it follows that consistency too is a syntactic concept, and is defined ultimately in terms of grammatical form alone. The inference rules of axiom systems, moreover, are always chosen so that derivability and consistency are interdefinable: that is, $p_1, \dots, p_n \vdash q$ if and only if $\{p_1, \dots, p_n, \neg q\}$ is inconsistent.

The semantic approach gives rise to *model theory*. Semantics studies the way sentences relate to “the world.” Truth is the central concept in semantics because a sentence is said to be true if it “corresponds to the world.” The concept of algebraic structure is used to make precise the meaning of “corresponds to the world.”

An *algebraic structure*, or simply *structure*, consists of a non-empty set of objects existing in the world w , called the domain and denoted below by D , and a function, called an *interpretation* and denoted below by R , that assigns to each constant an entity in D , to each predicate a relation among entities in D , and to each functor a function among entities in D . A sentence p is said to be true in w if the entities chosen as the interpretations of the sentence’s terms and functors *stand* to the relations chosen as the interpretation of the sentence’s predicates. We denote a structure by $\langle D, R \rangle$. Below we sometimes use \mathcal{A} to stand for $\langle D, R \rangle$ by writing $\mathcal{A} = \langle D, R \rangle$. A more traditional, algebraic notation for structure is used in Section 1.6. We will speak of *formulas* instead of sentences to allow for the possibility that a sentence contains free variables. The customary notation is to use $\mathcal{A} \models p$ to say p is true in the structure \mathcal{A} .

The semantic versions of validity and consistency are defined in terms of the concept of structure. In model theory validity is just called *validity*. Intuitively, an argument is valid if whenever the premises are true, so is the conclusion. More precisely, the argument from p_1, \dots, p_n to q is *valid* (in symbols, $p_1, \dots, p_n \models q$) if and only if, for all structures \mathcal{A} , if $\mathcal{A} \models p_1, \dots, \mathcal{A} \models p_n$, then $\mathcal{A} \models q$.

We are now ready to encounter, for the first time, *satisfiability*, the central concept of this handbook. Satisfiability is the semantic version of consistency. A set of formulas is said to be satisfiable if there is some structure in which all its component formulas are true: that is, $\{p_1, \dots, p_n\}$ is *satisfiable* if and only if, for some \mathcal{A} , $\mathcal{A} \models p_1$ and \dots and $\mathcal{A} \models p_n$. It follows from the definitions that validity and satisfiability are mutually definable: $p_1, \dots, p_n \models q$ if and only if $\{p_1, \dots, p_n, \neg q\}$ is unsatisfiable.

Although the syntactic and semantic versions of validity and consistency - namely derivability and consistency, on the one hand, and validity and satisfiability, on the other - have different kinds of definitions, the concepts from the two

branches of logic are systematically related. As will be seen later, for the languages studied in logic it is possible to devise axiom systems in such a way that the syntactic and semantic concepts match up so as to coincide exactly. Derivability coincides with validity (*i.e.* $p_1, \dots, p_n \vdash q$ if and only if $p_1, \dots, p_n \models q$), and consistency coincides with satisfiability (*i.e.* $\{p_1, \dots, p_n\}$ is consistent if and only if $\{p_1, \dots, p_n\}$ is satisfiable). Such an axiom system is said to be complete.

We have now located satisfiability, the subject of this handbook, in the broader geography made up of logic's basic ideas. Logic is about both validity and consistency, which are interdefinable in two different ways, one syntactic and one semantic. Among these another name for the semantic version of consistency is satisfiability. Moreover, when the language possesses a complete axiom system, as it normally does in logic, satisfiability also coincides exactly with syntactic consistency. Because of these correspondences, satisfiability may then be used to "characterize" validity (because $p_1, \dots, p_n \models q$ if and only if $\{p_1, \dots, p_n, \neg q\}$ is unsatisfiable) and derivability (because $p_1, \dots, p_n \vdash q$ if and only if $\{p_1, \dots, p_n, \neg q\}$ is unsatisfiable).

There is a further pair of basic logical ideas closely related to satisfiability: *necessity* and *possibility*. Traditionally, a sentence is said to be necessary (or necessarily true) if it is true in all possible worlds, and possible (or possibly true) if it is true in at least one possible world. If we understand a possible world to be a structure, possibility turns out to be just another name for satisfiability. A possible truth is just one that is satisfiable. In logic, the technical name for a necessary formula is logical truth: p is defined to be a *logical truth* (in symbols, $\models p$) if and only if, for all \mathcal{A} , $\mathcal{A} \models p$. (In sentential logic a logical truth is called a *tautology*.) Moreover, *necessary* and *possible* are predicates of the metalanguage (the language of logical theory) because they are used to describe sentences in the "object" language (the language that refers to entities in the world that is the object of investigation in logical theory).

There is one further twist. In the concept of consistency we have already the syntactic version of satisfiability. There is also a syntactic version of a logical truth, namely a theorem-in-an-axiom-system. We say p is a theorem of the system (in symbols $\vdash p$) if and only if p is derivable from the axioms alone. In a complete system, theorem-hood and logical truth coincide: $\vdash p$ if and only if $\models p$. Thus, in logical truth and theorem-hood we encounter yet another pair of syntactic and semantic concepts that, although they have quite different sorts of definitions, nevertheless coincide exactly. Moreover, a formula is necessary if it is not possibly not true. In other words, $\models p$ if and only if it is not the case that p is unsatisfiable. Therefore, satisfiability, theorem-hood, logical truths and necessities are mutually "characterizable."

This review shows how closely related satisfiability is to the central concepts of logic. Indeed, relative to a complete axiom system, satisfiability may be used to define, and may be defined by, the other basic concepts of the field - validity, derivability, consistency, necessity, possibility, logical truth, tautology, and theorem-hood.

However, although we have taken the trouble to clearly delineate the distinction between syntax and semantics in this section, it took over 2000 years before this was clearly enunciated by Tarski in the 1930s. Therefore, the formal notion

of satisfiability was absent until then, even though it was informally understood since Aristotle.

The early history of satisfiability, which will be sketched in the next sections, is the story of the gradual enrichment of languages from very simple languages that talk about crude physical objects and their properties, to quite sophisticated languages that can describe the properties of complex structures and computer programs. For all of these languages, the core concepts of logic apply. They all have a syntax with constants that stand for entities and with predicates that stand for relations. They all have sentences that are true or false relative to possible worlds. They all have arguments that are valid or invalid. They all have logical truths that hold in every structure. They all have sets that are satisfiable and others that are unsatisfiable. For all of them, logicians have attempted to devise complete axiom systems to provide syntactic equivalents that capture, in the set of theorems, exactly the set of logical truths, that replicate in syntactic derivations exactly the valid arguments, and provide derivations of contradictions from every unsatisfiable set. We shall even find examples in these early systems of attempts to define decision procedures for logical concepts. As we shall see, in all these efforts the concept of satisfiability is central.

1.2. The ancients

It was in Athens that logic as a science was invented by Aristotle (384-322 B.C.). In a series of books called the *Organon*, he laid the foundation that was to guide the field for the next 2000 years. The logical system he invented, which is called the syllogistic or *categorical* logic, uses a simple syntax limited to subject-predicate sentences.

Aristotle and his followers viewed language as expressing ideas that signify entities and the properties they instantiate in the “world” outside the mind. They believed that concepts are combined to “form” subject-predicate propositions in the mind. A mental thought of a proposition was something like being conscious of two concepts at once, the subject S and the predicate P . Aristotle proposed four different ways to capture this notion of thought, with respect to a given “world” w , depending on whether we link the subject and predicate universally, particularly, positively, or negatively: that is, every S is P , no S is P , some S is P , and some S is not P . These *categorical propositions* were called, respectively, A (universal affirmative), E (universal negative), I (particular affirmative), and O (particular negative) propositions. Their truth-conditions are defined as follows:

A :	Every S is P is true in w	iff	Everything in w signified by S is something signified in w by S and P
E :	No S is P is true in w	iff	Some S is P is false in w
I :	Some S is P is true in w	iff	There is some T such that everything signified in w by S and P is something that is signified in w by P and T

O : Some S is not P iff Every S is P is false in w
 is true in w

These propositions have the following counterparts in set theory:

$S \subseteq P$	iff	$S = S \cap P$
$S \cap P = \emptyset$	iff	$\neg(S \cap P \neq \emptyset)$
$S \cap P \neq \emptyset$	iff	$\exists T : S \cap P = P \cap T$
$S \cap \bar{P} \neq \emptyset$	iff	$\neg(S = S \cap P)$

The resulting logic is two-valued: every proposition is true or false. It is true that Aristotle doubted the universality of this bivalence. In a famous discussion of so-called future contingent sentences, such as “there will be a sea battle tomorrow,” he pointed out that a sentence like this, which is in the future tense, is not now determined to be either true or false. In modern terms such a sentence “lacks a truth-value” or receives a third truth-value. In classical logic, however, the law of excluded middle (commonly known as *tertium non datur*), that is, p or not p is always true, was always assumed.

Unlike modern logicians who accept the empty set, classical logicians assumed, as a condition for truth, that a proposition’s concepts must signify at least one existing thing. Thus, the definitions above work only if T is a non-empty set. It follows that A and E propositions cannot both be true (they are called contraries), and that I and O propositions cannot both be false. The definitions are formulated in terms of identity because doing so allowed logicians to think of mental proposition formulation as a process of one-to-one concept comparison, a task that consciousness seemed perfectly capable of doing.

In this theory of mental language we have encountered the first theory of satisfiability. A proposition is satisfiable (or possible, as traditional logicians would say) if there is some world in which it is true. A consistent proposition is one that is satisfiable. Some propositions were recognized as necessary, or always true, for example: every S is S .

Satisfiability can be used to show that propositions p_1, \dots, p_n do not logically imply q : one only needs to show that there is some assignment of concepts to the terms so that all the propositions in $\{p_1, \dots, p_n, \neg q\}$ come out true. For example, consider the statement:

$$(\text{some } M \text{ is } A \wedge \text{some } C \text{ is } A) \rightarrow \text{every } M \text{ is } C.$$

Aristotle would show this statement is false by replacing the letters with familiar terms to obtain the requisite truth values: some man is an animal and some cow is an animal are both true, but every man is a cow is false. In modern terms, we say the set

$$\{ \text{some } M \text{ is } A, \text{some } C \text{ is } A, \neg(\text{every } M \text{ is } C) \}$$

is satisfiable.

The means to deduce (that is, provide a valid argument) was built upon syllogisms, using what is essentially a complete axiom system for any conditional $(p_1, \dots, p_n) \rightarrow q$ in which p_1, \dots, p_n , and q are categorical propositions [Mar97].

A syllogism is defined as a conditional $(p \wedge q) \rightarrow r$ in which p, q , and r are A, E, I , or O propositions. To show that propositions are valid, that is $(p_1 \wedge \dots \wedge p_n) \rightarrow q$, Aristotle would create syllogisms $(p_1 \wedge p_2) \rightarrow r_1, (r_1 \wedge p_3) \rightarrow r_2, \dots, (r_{n-2} \wedge p_n) \rightarrow q$, then repeatedly reduce valid syllogisms to one of

- A1: $(\text{every } X \text{ is } Y \wedge \text{every } Y \text{ is } Z) \rightarrow \text{every } X \text{ is } Z$
 A2: $(\text{every } X \text{ is } Y \wedge \text{no } Y \text{ is } Z) \rightarrow \text{no } X \text{ is } Z$

The reduction, when viewed in reverse, is an axiom system where A1 and A2 are axiom schemata, from which are deduced the valid syllogisms, and from the valid syllogisms are deduced all valid conditionals. The system used four inference rules:

- R1: From $(p \wedge q) \rightarrow r$ infer $(\neg r \wedge q) \rightarrow \neg p$
 R2: From $(p \wedge q) \rightarrow r$ infer $(q \wedge p) \rightarrow r$
 R3: From no X is Y infer no Y is X
 R4: From $(p \wedge q) \rightarrow \text{no } X \text{ is } Y$ infer $(p \wedge q) \rightarrow \text{some } X \text{ is not } Y$

For example, to prove

$$(\text{every } P \text{ is } M \wedge \text{no } S \text{ is } M) \rightarrow \text{some } S \text{ is not } P$$

one would deduce

1. $(\text{every } P \text{ is } M \wedge \text{no } M \text{ is } S) \rightarrow \text{no } P \text{ is } S$ Axiom A2
2. $(\text{every } P \text{ is } M \wedge \text{no } S \text{ is } M) \rightarrow \text{no } S \text{ is } P$ Rule R3
3. $(\text{every } P \text{ is } M \wedge \text{no } S \text{ is } M) \rightarrow \text{some } S \text{ is not } P$ Rule R4

The logic of the Stoics (c. 300s-200s BC) developed into a sophisticated sentential logic, using operators $\rightarrow, \wedge, \neg$, and \vee , where a proposition is the *meaning* of a sentence that expresses it and the truth of a proposition may change over time. They combined this with the standard definition of validity to discover a series of propositional inferences that have remained part of logical lore ever since:

$$\begin{array}{ll} p, p \rightarrow q \models q & (\text{modus ponens}) \\ \neg q, p \rightarrow q \models \neg p & (\text{modus tollens}) \\ \neg q, p \vee q \models p & (\text{disjunctive syllogism}) \\ p \rightarrow q, q \rightarrow r \models p \rightarrow r & (\text{hypothetical syllogism}) \end{array}$$

1.3. The medieval period

Logicians of the medieval period knew all of the logic of Aristotle and the Stoics, and much more. The syntax of the languages they used was rich, incorporating combined categorical propositions (with and without modal and epistemic operators), other quantifiers, restrictive and non-restrictive relative clauses, and the propositional connectives into complex sentences. Moreover, although they did

not have set theory, they described interpretations of predicates using set-like notions such as “group” or “collection.”

The development of concepts open to decision by an effective process (such as a *mechanical process*) was actually an important goal of early modern logic, although it was not formulated in those terms. A goal of symbolic logic is to make epistemically transparent judgments that a formula is a theorem of an axiom system or is deducible within an axiom system. An effective process ensures this transparency because it is possible to know with relative certainty that each stage in the process is properly carried out.

The work of Ramon Lull (1232-1315) was influential beyond the medieval period. He devised the first system of logic based on diagrams expressing truth and rotating wheels to achieve some form of deduction. It had similarities to important later work, for example Venn circles, and greatly influenced Leibniz in his quest for a system of deduction that would be universal.

1.4. The renaissance

In the 17th century Descartes and Leibniz began to understand the power of applying algebra to scientific reasoning. To this end, Leibniz devised a language that could be used to talk about either ideas or the world. He thought, like we do, that sets stand to one another in the subset relation \subseteq , and that a new set can be formed by intersection \cap . He also thought that concepts can be combined by definitions: for example the concepts *animal* and *rational* can be combined to form the concept *rational+animal*, which is the definition of the concept *man*, and the concept animal would then be a “part” of the concept *man*. The operator \preceq , called concept inclusion, was introduced to express this notion: thus, animal \preceq man.

Leibniz worked out dual Boolean interpretations of syllogistic propositions joined with the propositional connectives. The first (*intensional*) interpretation assigns terms to “concepts” within a structure of concepts ordered by \preceq and organized by operations that we would call *meet* and *join*. The dual (*extensional*) interpretation is over a Boolean algebra of sets.

The logical operations of multiplication, addition, negation, identity, class inclusion, and the null class were known at this time, well before Boole, but Leibniz published nothing on his ideas related to formal logic. In addition to \preceq his logic is rooted in the operators of identity ($=$), and a conjunction-like operator (\oplus) called *real addition*, which obeys the following:

$$\begin{array}{ll} t \oplus t = t & \text{(idempotency)} \\ t \oplus t' = t' \oplus t & \text{(commutativity)} \\ t \oplus (t' \oplus t'') = (t \oplus t') \oplus t'' & \text{(associativity)} \end{array}$$

where t , t' , and t'' are terms representing substance or ideas. The following, Leibniz’s equivalence, shows the tie between set inclusion and real addition that is the basis of his logics.

$$t \preceq t' \text{ if and only if } t \oplus t' = t'$$

Although Leibniz's system is simplistic and ultimately implausible as an account of science, he came very close to defining with modern rigor complete axiom systems for well-defined formal languages that also possessed decision procedures for identifying the sentences satisfied in every interpretation. It would be 250 years before modern logic accomplished the same for its more complex languages. His vision is relevant to modern times in other ways too. As examples, he invented binary arithmetic, and his *calculus ratiocinator* is regarded by some as a formal inference engine, its use not unlike that of a computer programming language, and by others as referring to a "calculating machine" that was a forerunner to the modern digital computer. In fact, Leibniz constructed a machine, called a Stepped Reckoner, for mathematical calculations. Yet, at this point in history, the notion of satisfiability still had not been enunciated.

1.5. The first logic machine

According to Gardner [Gar58] the first *logic machine*, that is the first machine able to solve problems in formal logic (in this case syllogisms), was invented by Charles Stanhope, 3rd Earl Stanhope (1753-1816). It employed methods similar to Venn circles (Section 1.6) and therefore can in some sense be regarded as Boolean. However, it was also able to go beyond traditional syllogisms to solve numerical syllogisms such as the following: 8 of 10 pigeons are in holes, and 4 of the 10 pigeons are white (conclude at least 2 holes have white pigeons). See [Gar58] for details.

1.6. Boolean algebra

George Boole (1815-1864) advanced the state of logic considerably with the introduction of the algebraic structure that bears his name¹: a structure $\langle B, \vee, \wedge, \neg, 0, 1 \rangle$ is a Boolean algebra if and only if \vee and \wedge are binary operations and \neg is a unary operation on B under which B is closed, $1, 0 \in B$, and

$$\begin{array}{ll} x \wedge y = y \wedge x; & x \vee \neg x = 1; \\ x \vee y = y \vee x; & 1 \wedge x = x; \\ x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z); & 0 \vee x = x; \\ x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z); & x \wedge \neg x = 0; \end{array}$$

Boole's main innovation was to develop an algebra-like notation for the elementary properties of sets. His own objective, at which he succeeded, was to provide a more general theory of the logic of terms which has the traditional syllogistic logic as a special case. He was one of the first to employ a symbolic language. His notation consisted of term variables s, t, u, v, w, x, y, z etc., which he interpreted as standing for sets, the standard "Boolean" operators on sets (complementation indicated by \neg , union indicated by \vee , intersection indicated by \wedge), constants for the universal and empty set (1 and 0), and the identity sign ($=$) used to form equations. He formulated many of the standard "laws" of Boolean algebra, including association, commutation, and distributions, and noted many of

¹The structures we call 'Boolean algebras' were not defined by Boole but by Jevons (see below) who advocated the use of the inclusive or.

the properties of complementation and the universal and empty sets. He symbolized Aristotle's four categorical propositions, relative to subject y and predicate x , by giving names of convenience V ($V \neq 0$) to sets that intersect appropriately with y and x forming subsets:

	Boole	Sets
Every x is y :	$x = x \cdot y$	$x \subseteq y$
No x is y :	$0 = x \cdot y$	$x \subseteq \bar{y}$
Some x is y :	$V = V \cdot x \cdot y$	$x \cap y \neq \emptyset$
Some x is not y :	$V = V \cdot x \cdot (1 - y)$	$x \cap \bar{y} \neq \emptyset$

Boole was not interested in axiom systems, but in the theory of inference. In his system it could be shown that the argument from p_1, \dots, p_n to q is valid (*i.e.* $p_1, \dots, p_n \models q$) by deriving the equation q from equations p_1, \dots, p_n by applying rules of inference, which were essentially cases of algebraic substitution.

However, Boole's algebras needed a boost to advance their acceptance. Gardner [Gar58] cites John Venn (1834-1923) and William Stanley Jevons (1835-1882) as two significant contributors to this task. Jevons regarded Boole's work as the greatest advance since Aristotle but saw some flaws that he believed kept it from significantly influencing logicians of the day, particularly that it was too mathematical. To fix this he introduced, in the 1860's, the "method of indirect inference" which is an application of *reductio ad absurdum* to Boolean logic. For example, to handle 'All x is y ' and 'No y is z ' Jevons would write all possible "class explanations" as triples

$$xyz, xy\bar{z}, x\bar{y}z, x\bar{y}\bar{z}, \bar{x}yz, \bar{x}y\bar{z}, \bar{x}\bar{y}z, \bar{x}\bar{y}\bar{z},$$

where the first represents objects in classes x , y , and z , the second represents objects in classes x and y but not in z , and so on². Then some of these triples would be eliminated by the premises which imply they are empty. Thus, $x\bar{y}z$ and $x\bar{y}\bar{z}$ are eliminated by 'All x is y ' and xyz and $\bar{x}yz$ are eliminated by 'No y is z '. Since no remaining triples contain both x and z , one can conclude 'No x is z ' and 'No z is x '.

Although Jevons' system is powerful it has problems with some statements: for example, 'Some x is y ' cannot eliminate any of the triples xyz and $xy\bar{z}$ since at least one but maybe both represent valid explanations. Another problem is the exponential growth of terms, although this did not seem to be a problem at the time³. Although the reader can see x , y and z taking truth values 0 or 1, Jevons did not appreciate the fact that truth-value logic would eventually replace class logic and his attention was given only to the latter. He did build a logic machine (called a "logic piano" due to its use of keys) that can solve five term problems [Jev70] (1870). The reader is referred to [Gar58] for details.

What really brought clarity to Boolean logic, though, was the contribution of Venn [Ven80] (1880) of which the reader is almost certainly acquainted since it touches several fields beyond logic. We quote the elegantly stated passage of [Gar58] explaining the importance of this work:

²Jevons did not use $\bar{}$ but lower and upper case letters to distinguish inclusion and exclusion.

³Nevertheless, Jevons came up with some labor saving devices such as inked rubber stamps to avoid having to list all possibilities at the outset of a problem.

It was of course the development of an adequate symbolic notation that reduced the syllogism to triviality and rendered obsolete all the quasi-syllogisms that had been so painfully and exhaustively analyzed by the 19th century logicians. At the same time many a controversy that once seemed so important no longer seemed so. ... Perhaps one reason why these old issues faded so quickly was that, shortly after Boole laid the foundations for an algebraic notation, John Venn came forth with an ingenious improvement to Euler's circles⁴. The result was a diagrammatic method so perfectly isomorphic with the Boolean class algebra, and picturing the structure of class logic with such visual clarity, that even a nonmathematically minded philosopher could "see" what the new logic was all about.

As an example, draw a circle each for x , y , and z and overlap them so all possible intersections are visible. A point inside circle x corresponds to x and a point outside circle x corresponds to $\neg x$ and so on. Thus a point inside circles x and y but outside circle z corresponds to Jevons' $xy\neg z$ triple. Reasoning about syllogisms follows Jevons as above except that the case 'Some x is y ' is handled by placing a mark on the z circle in the region representing xy which means it is not known whether the region xyz or $xy\neg z$ contains the x that is y . Then, if the next premise is, say, 'All y is z ', the $xy\neg z$ region is eliminated so the mark moves to the xyz region it borders allowing the conclusion 'Some z is x '. Since the connection between 0-1 logic and Venn circles is obvious, syllogisms can be seen as just a special case of 0-1 logic.

The results of Boole, Jevons, and Venn rang down the curtain on Aristotelian syllogisms, ending a reign of over 2000 years. In the remainder of this chapter syllogisms will appear only once, and that is to show cases where they are unable to represent common inferences.

1.7. Frege, logicism, and quantification logic

In the nineteenth century mathematicians like Cauchy and Weierstrass put analysis on a clear mathematical footing by precisely defining its central terms. George Cantor (1845-1918) extended their work by formulating a more global theory of sets that allowed for the precise specification of such important details as when sets exist, when they are identical, and what their cardinality is. Cantor's set theory was still largely intuitive and imprecise, though mathematicians like Dedekind and Peano had axiomatized parts of number theory. Motivated as much by philosophy as logic, the mathematician Gottlob Frege (1848-1925) conceived a project called logicism to deduce from the laws of logic alone "all of mathematics," by which he meant set theory, number theory, and analysis [Lav94]. Logicism as an attempt to deduce arithmetic from the axioms of logic was ultimately a failure. As a research paradigm, however, it made the great contribution of making clear the boundaries of "formal reasoning," and has allowed deep questions to be posed, but which are still unanswered, about the nature of mathematical truth.

Frege employed a formal language of his own invention, called "concept notation" (Begriffsschrift). In it he invented a syntax which is essential to the formal

⁴The difference between Euler circles and Venn circles is that Venn circles show all possible overlaps of classes while there is no such requirement for Euler circles. Therefore, Euler circles cannot readily be used as a means to visualize reasoning in Boolean logic. Leibniz and Ramon Lull also used Euler-like circles [Bar69].

language we know today for sentential and quantificational logic, and for functions in mathematics. In his *Grundgesetze der Arithmetik* (1890) he used a limited number of “logical axioms,” which consisted of five basic laws of sentential logic, several laws of quantifiers, and several axioms for functions. When he published this work, he believed that he had succeeded in defining within his syntax the basic concepts of number theory and in deducing their fundamental laws from his axioms of logic alone.

1.8. Russell and Whitehead

Bertrand Russell (1882-1970), however, discovered that he could prove in Frege’s system his notorious paradox: if the set of all sets that are not members of themselves is a member of itself, then, by definition, it must not be a member of itself; and if it is not a member of itself then, by definition, it must be a member of itself.

In *Principia Mathematica* (1910-13), perhaps the most important work of early modern logic, Russell and Whitehead simplified Frege’s logical axioms of functions by substituting for a more abstract set describing sets and relations. One of the great weaknesses of traditional logic had been its inability to represent inferences that depend on relations. For example, in the proof of Proposition 1 of his *Elements* Euclid argues, regarding lines, that if $AC=AB$ and $BC=AB$, it follows by commutativity that $CA=AB$, and hence that $AC=BC$ (by either the substitutivity or transitivity or identity). But, the requirement of Aristotelian logic that all propositions take subject-predicate form ‘ S is P ’ makes it impossible to represent important grammatical facts: for example, that in a proposition $AC = AB$, the subject AC and direct object AB are phrases made up of component terms A , B , and C , and the fact that the verb $=$ is transitive and takes the direct object AB . The typical move was to read “ $=AC$ ” as a unified predicate and to recast equations in subject-predicate form:

$AC = AB$:	the individual AC has the property being-identical-to- AB
$BC = AB$:	the individual BC has the property being-identical-to- AB
$AC = BC$:	the individual AC has the property being-identical-to- BC

But the third line does not follow by syllogistic logic from the first two.

One of the strange outcomes of early logic is that syllogistic logic was so unsuccessful at representing mathematical reasoning that in the 2000 years in which it reigned there was only one attempt to reproduce geometrical proofs using syllogisms; it did not occur until the 16th century, and it was unsuccessful [HD66]. Boole’s logic shared with Aristotle’s an inability to represent relations. Russell, however, was well aware both of the utility of relations in mathematics and the need to reform syntax to allow for their expression [Rus02].

Accordingly, the basic notation of *Principia* allows for variables, one-place predicates standing for sets, and n -place predicates standing for relations. Formulas were composed using the sentential connectives and quantifiers. With just this notation it was possible to define all the constants and functions necessary for arithmetic, and to prove with simplified axioms a substantial part of number

theory. It appeared at first that *Principia* had vindicated logicism: arithmetic seemed to follow from logic.

1.9. Gödel's incompleteness theorem

In 1931, Kurt Gödel astounded the mathematical world by proving that the axiom system of *Principia*, and indeed any axiom system capable of formulating the laws of arithmetic, is and must be incomplete in the sense that there will always be some truth of arithmetic that is not a theorem of the system. Gödel proved, in short, that logicism is false - that mathematics in its entirety cannot be deduced from logic. Gödel's result is sweeping in its generality. It remains true, however, that limited parts of mathematics are axiomatizable, for example first-order logic. It is also true that, although incomplete, mathematicians as a rule still stick to axiomatic formulations as their fundamental methodology even for subjects that they know must be incomplete. Axiomatic set theory, for example, contains arithmetic as a part and is therefore provably incomplete. But axiomatics is still the way set theory is studied even within its provable limitations.

Although logicism proved to be false, the project placed logic on its modern foundations: *Principia* standardized the modern notation for sentential and quantificational logic. In the 1930s Hilbert and Bernays christened "first-order" logic as that part of the syntax in which quantifiers bind only variables over individuals, and "higher-order" logic as the syntax in which variables bind predicates, and predicates of predicates, etc.

1.10. Effective process and recursive functions

As logic developed in the 20th century, the importance of effective decidability (see Section 1.3) increased and defining effective decidability itself became a research goal. However, since defining any concept that incorporates as a defining term "epistemic transparency" would require a background theory that explained "knowledge," any direct definition of effective process remained elusive. This difficulty motivated the inductive definitions of recursive function theory, Turing machines, lambda calculus, and more, which, by Church's thesis, succeeds in providing an indirect definition of effective process.

One outcome of logicism was the clarification of some of the basic concepts of computer science: for example, the notion of recursive functions which are supposed to be what a mathematician would intuitively recognize as a "calculation" on the natural numbers. Gödel, in his incompleteness paper of 1931, gave a precise formal definition of the class of primitive recursive functions, which he called "recursive functions." Gödel first singled out three types of functions that all mathematicians agree are calculations and called these recursive functions. He then distinguished three methods of defining new functions from old. These methods moreover were such that everyone agreed they produced a calculation as an output if given calculations as inputs. He then defined the set of recursive functions as the closure of the three basic function types under the three construction methods.

Given the definition of recursive function, Gödel continued his incompleteness proof by showing that for each calculable function, there is a predicate in the language of *Principia* that has that function as its extension. Using this fact, he then showed, in particular, that *Principia* had a predicate T that had as its extension the theorems of *Principia*. In an additional step he showed that *Principia* also possesses a constant c that stands for the so-called liar sentence $\neg Tc$, which says in effect “This sentence is not a theorem.” Finally, he demonstrated both that $\neg Tc$ is a truth of arithmetic and that it is not a theorem of *Principia*. He proved, therefore, that the axiom system failed to capture one of the truths of arithmetic and was therefore incomplete.

Crucial to the proof was its initial definition of recursive function. Indeed, by successfully analyzing “effective process,” Gödel made a major contribution to theoretical computer science: the computable function. However, in his Princeton lectures of 1934, Gödel, attributing the idea of general recursive functions to a suggestion of Herbrand, did not commit himself to whether all effective functions are characterized by his definition. In 1936, Church [Chu36] and Turing [Tur36] independently proposed a definition of the effectively computable functions. Its equivalence with Gödel’s definition was proved in 1943 by Kleene [Kle43]. Emil Post (1897-1954), Andrei Markov (1903-1979), and others confirmed Gödel’s work by providing alternative analyses of computable function that are also provably coextensive with his.

1.11. Herbrand’s theorem

Herbrand’s theorem relates issues of validity and logical truth into ones of satisfiability, and issues of satisfiability into ones concerning the definition of computable functions on syntactic domains. The proof employs techniques important to computer science. A *Herbrand model* for a formula of first-order logic has as its domain literally those terms generated from terms that occur in the formula p . Moreover, the predicates of the model are true of a term if and only if the formula asserting that the predicate holds of the term occurs in p . The first part of Herbrand’s theorem says that p is satisfiable if and only if it is satisfied in its Herbrand model.

Using techniques devised by Skolem, Herbrand showed that the quantified formula p is satisfiable if and only if a specific set of its truth-functional instantiations, each essentially a formula in sentential logic, is satisfiable. Thus, satisfiability of p reduces to an issue of testing by truth-tables the satisfiability of a potentially infinite set S of sentential formulas. Herbrand showed that, for any first-order formula p , there is a decision function f such that $f(p) = 1$ if p is unsatisfiable because, eventually, one of the truth-functions in S will come out false in a truth-table test, but $f(p)$ may be undefined when p is satisfiable because the truth-table testing of the infinite set S may never terminate.

1.12. Model theory and Satisfiability

Although ideas in semantics were central to logic in this period, the primary framework in which logic was studied was the axiom system. But the seemingly

obvious need to define the grammar rules for a formal language was skipped over until Gödel gave a completely formal definition of the formulas of his version of simple type theory in his 1931 paper [Mel92].

In the late nineteenth and early twentieth centuries Charles Sanders Peirce, see [Lew60], and Ludwig Wittgenstein [Wit33] had employed the two-valued truth-tables for sentential logic. In 1936 Marshall Stone proved that the more general class of Boolean algebras is of fundamental importance for the interpretation of classical logic. His “representation theorem” showed that any interpretation of sentential logic that assigns to the connectives the corresponding Boolean operators and defines validity as preserving a “designated value” defined as a maximal upwardly closed subset of elements of the algebra (called a “filter”) has as its logical truths and valid arguments exactly those of classical logic [Sto36]. The early decades of the twentieth century also saw the development of many-valued logic, in an early form by Peirce [TTT66] and then in more developed versions by Polish logicians lead by Jan Łukasiewicz (1878-1956). Thus, in sentential logic the idea of satisfiability was well understood as “truth relative to an assignment of truth-values” in a so-called “logical matrix” of truth-functions.

A precise notion of satisfiability for first-order logic, however, was not developed until the 1930s in the work of Alfred Tarski (1902-1983) [Tar56, Tar44, Tar54]. Tarski’s task was to define a set of necessary and sufficient conditions for “ p is true,” for any formula p of first-order syntax. His solution was not to define the idea in a single phrase applicable to all formulas, but, like Gödel, to give an inductive definition, first defining *truth* for basic formulas and then extending the definition to more complex formulas. The problem was made complex, however, by the fact that, unlike sentential logic in which the truth-value of the parts immediately determine that of the whole (by reference to truth-tables), when the whole expression is universally quantified, it is unclear how its truth is determined by the interpretation of its part. How does the “truth” of the open formula Fx determine that of $\forall x : Fx$?

Tarski solved the problem in two stages. In the first stage he assigned fixed interpretations to the variables. Having done so, it is possible to say when $\forall x : Fx$ is true if we know whether Fx is true under its various interpretations. If Fx is true under all interpretation of x , then $\forall x : Fx$ is also true under each of these interpretations. If Fx is false under even one interpretation of x , however, $\forall x : Fx$ is false under any interpretation of the variables. Tarski coined the technical term *satisfaction* to refer to truth relative to an interpretation of variables.

Let $\mathcal{A} = \langle D, R \rangle$ be a structure and define a variable assignment as any function s that assigns to each variable an entity in D . Given R and s , all the basic expressions of the syntax have a referent relative to D . We can now inductively define “ p is satisfied relative to \mathcal{A} and s .” The atomic formula Ft_1, \dots, t_n is satisfied relative to \mathcal{A} and s if and only if the interpretations of t_1, \dots, t_n in R and s stand in the relation assigned by R to F . The “satisfaction” of molecular formulas made up by the sentential connectives are determined by the two-valued truth-tables depending on whether or not their parts are satisfied. Finally, $\forall x : Fx$ is satisfied relative to \mathcal{A} and s if and only if Fx is satisfied relative to R and every variable assignment s . The notation for “ p is satisfied relative to \mathcal{A} and s ” is $\mathcal{A} \models_s p$.

The second stage of Tarski's definition is to abstract away from a variable assignment and define the simpler notion "p is true relative to \mathcal{A} ." His idea at this stage is to interpret an open formula Fx as true in this general sense if it is "always true" in the sense of being satisfied under every interpretation of its variables. That is, he adopts the simple formula: p is true relative to \mathcal{A} if and only if, for all variable assignments s , p is satisfied relative to \mathcal{A} and s . In formal notation, $\mathcal{A} \models p$ if and only if, for all s , $\mathcal{A} \models_s p$.

Logicians have adopted the common practice of using the term "satisfied in a structure" to mean what Tarski called "true in a structure." Thus, it is common to say that p is satisfiable if there is some structure \mathcal{A} such that p is true in \mathcal{A} , and that a set of formulas X is satisfiable if and only if there is some structure \mathcal{A} such that for all formulas p in X , p is true (satisfied) in \mathcal{A} .

1.13. Completeness of first-order logic

First-order logic has sufficient expressive power for the formalization of virtually all of mathematics. To use it requires a sufficiently powerful axiom system such as the Zermelo-Fraenkel set theory with the axiom of choice (ZFC). It is generally accepted that all of classical mathematics can be formalized in ZFC.

Proofs of completeness of first-order logic under suitable axiom systems date back at least to Gödel in 1929. In this context, completeness means that all logically valid formulas of first-order logic can be derived from axioms and rules of the underlying axiom system. This is not to be confused with Gödel's incompleteness theorem which states that there is no consistent axiom system for the larger set of truths of number theory (which includes the valid formulas of first-order logic as a proper subset) because it will fail to include at least one truth of arithmetic.

Tarski's notion of truth in a structure introduced greater precision. It was then possible to give an elegant proof that first-order logic is complete under its usual axiom systems and sets of inference rules. Of particular interest is the proof due to Leon Henkin (1921-2006) that makes use of two ideas relevant to this book: satisfiability and a structure composed of syntactic elements [Hen49] (1949). Due to the relationship of validity to satisfiability, Henkin reformulated the difficult part of the theorem as: if a set of formulas is consistent, then it is satisfiable. He proved this by first extending a consistent set to what he calls a maximally consistent saturated set, and then showing that this set is satisfiable in a structure made up from the syntactic elements of the set. Although differing in detail, the construction of the structure is similar to that of Herbrand.

Herbrand models and Henkin's maximally consistent saturated sets are relevant prototypes of the technique of constructing syntactic proxies for conventional models. In complexity theory, the truth of predicates is typically determined relative to a structure with a domain of entities that are *programs* or *languages* which themselves have semantic content that allow one to determine a corresponding, more conventional, model theoretic structure. In that sense, the program or language entities can be said to be a proxy for the conventional model and the predicates are second-order, standing for a property of sets.

1.14. Application of logic to circuits

Claude Shannon provided one of the bridges connecting the path of logic over the centuries to its practical applications in the information and digital age. Another bridge is considered in the next section. Whereas the study of logic for thousands of years was motivated by a desire to explain how humans use information and knowledge to deduce facts, particularly to assist in decision making, Shannon, as a student at MIT, saw propositional logic as an opportunity to make rigorous the design and simplification of switching circuits. Boolean algebra, appreciated by a relatively small group of people for decades, was ready to be applied to the fledgling field of digital computers.

In his master's thesis [Sha40] (1940), said by Howard Gardner of Harvard University to be "possibly the most important, and also the most famous, master's thesis of the century," Shannon applied Boolean logic to the design of minimal circuits involving relays. Relays are simple switches that are either "closed," in which case current is flowing through the switch or "open," in which case current is stopped. Shannon represented the state of a relay with a variable taking value 1 if open and the value 0 if closed. His algebra used the operator '+' for "or" (addition - to express the state of relays in series), '.' for "and" (multiplication - to express the state of relays in parallel), and his notation for the negation of variable x was x' .

The rigorous synthesis of relay circuits entails expressing and simplifying complex Boolean functions of many variables. To support both goals Shannon developed two series expansions for a function, analogous, in his words, to Taylor's expansion on differentiable functions. He started with

$$f(x_1, x_2, \dots, x_n) = x_1 \cdot f(1, x_2, \dots, x_n) + x_1' \cdot f(0, x_2, \dots, x_n)$$

which we recognize as the basic splitting operation of DPLL algorithms and the *Shannon expansion* which is the foundation for Binary Decision Diagrams (Section 1.20), and its dual

$$f(x_1, x_2, \dots, x_n) = (f(0, x_2, \dots, x_n) + x_1) \cdot (f(1, x_2, \dots, x_n) + x_1').$$

Using the above repeatedly he arrived at the familiar DNF canonical form

$$\begin{aligned} f(x_1, x_2, \dots, x_n) = & f(0, 0, \dots, 0) \cdot x_1' \cdot x_2' \cdot \dots \cdot x_n' + \\ & f(1, 0, \dots, 0) \cdot x_1 \cdot x_2' \cdot \dots \cdot x_n' + \\ & f(0, 1, \dots, 0) \cdot x_1' \cdot x_2 \cdot \dots \cdot x_n' + \\ & \dots \\ & f(1, 1, \dots, 1) \cdot x_1 \cdot x_2 \cdot \dots \cdot x_n \end{aligned}$$

and its familiar CNF dual. These support the expression of any relay circuit and, more generally, any combinational circuit. To simplify, he introduced the following operations:

$$\begin{aligned} x &= x + x = x + x + x = \dots \\ x + x \cdot y &= x \end{aligned}$$

$$\begin{aligned}
 x \cdot f(x) &= x \cdot f(1) \\
 x' \cdot f(x) &= x' \cdot f(0) \\
 x \cdot y + x' \cdot z &= x \cdot y + x' \cdot z + y \cdot z
 \end{aligned}$$

and their duals. In the last operation the term $y \cdot z$ is the *consensus* of terms $x \cdot y$ and $x' \cdot z$. The dual of the last operation amounts to adding a propositional *resolvent* to a CNF clause set. The first two operations are *subsumption* rules.

In his master's thesis Shannon stated that one can use the above rules to achieve minimal circuit representations but did not offer a systematic way to do so. Independently, according to Brown [Bro03], Archie Blake, an all but forgotten yet influential figure in Boolean reasoning, developed the notion of consensus in his Ph.D. thesis [Bla37] of 1937. Blake used consensus to express Boolean functions in a minimal DNF form with respect to their *prime implicants* (product g is an implicant of function h if $g \cdot h' = 0$ and is prime if it is minimal in literals) and subsumption. An important contribution of Blake was to show that DNFs are not minimized unless consensus is not possible. The notion of consensus was rediscovered by Samson and Mills [SM54] (1954) and Quine [Qui55] (1955). Later, Quine [Qui59] (1959) and McCluskey [McC59] (1959) provided a systematic method for the minimization of DNF expressions through the notion of *essential prime implicants* (necessary prime implicants) by turning the 2-level minimization problem into a covering problem. This was perhaps the first confrontation with complexity issues: although they did not know it at the time, the problem they were trying to solve is \mathcal{NP} -complete and the complexity of their algorithm was $O(3^n/\sqrt{n})$, where n is the number of variables.

1.15. Resolution

Meanwhile, the advent of computers stimulated the emergence of the field of automated deduction. Martin Davis has written a history of this period in [Dav01] on which we base this section. Early attempts at automated deduction were aimed at proving theorems from first-order logic because, as stated in Section 1.13, it is accepted that given appropriate axioms as premises, all reasoning of classical mathematics can be expressed in first-order logic. Propositional satisfiability testing was used to support that effort.

However, since the complexity of the problem and the space requirements of a solver were not appreciated at the time, there were several notable failures. At least some of these determined satisfiability either by simple truth table calculations or expansion into DNF and none could prove anything but the simplest theorems. But, each contributed something to the overall effort. According to Davis [Dav01], Gilmore's [Gil60] (1960) system served as a stimulus for others and Prawitz [Pra60] (1960) adopted a modified form of the method of semantic tableaux. Also notable were the "Logic Theory Machine" of [NSS57] (1957), which used the idea of a search heuristic, and the Geometry machine of [Gel59] (1959), which exploited symmetry to reduce proof size.

Things improved when Davis and Putnam proposed using CNF for satisfiability testing as early as 1958 in an unpublished manuscript for the NSA [DP58].

According to Davis [Dav01], that manuscript cited all the essentials of modern DPLL variants. These include:

1. The one literal rule also known as the unit-clause-rule: for each clause (l), called a *unit clause*, remove all clauses containing l and all literals $\neg l$.
2. The affirmative-negative rule also known as the pure-literal-rule: if literal l is in some clause but $\neg l$ is not, remove all clauses containing l . Literal l is called a *pure literal*.
3. The rule for eliminating atomic formulas: that is, replace

$$(v \vee l_{1,1} \vee \dots \vee l_{1,k_1}) \wedge (\neg v \vee l_{2,1} \vee \dots \vee l_{2,k_2}) \wedge C$$

with

$$(l_{1,1} \vee \dots \vee l_{1,k_1} \vee l_{2,1} \vee \dots \vee l_{2,k_2}) \wedge C$$

if literals $l_{1,i}$ and $l_{2,j}$ are not complementary for any i, j .

4. The splitting rule, called in the manuscript ‘the rule of case analysis.’

Observe that rule 3. is ground resolution: the CNF expression it is applied to having come from a prenex form with its clauses grounded. The published version of this manuscript is the often cited [DP60] (1960). The Davis-Putnam procedure, or DPP, reduced the size of the search space considerably by eliminating variables from a given expression. This was done by repeatedly choosing a target variable v still appearing in the expression, applying all possible ground resolutions on v , then eliminating all remaining clauses still containing v or $\neg v$.

Loveland and Logemann attempted to implement DPP but they found that ground resolution used too much RAM, which was quite limited in those days. So they changed the way variables are eliminated by employing the splitting rule: recursively assigning values 0 and 1 to a variable and solving both resulting subproblems [DLL62] (1962). Their algorithm is, of course, the familiar DPLL.

Robinson also experimented with DPP and, taking ideas from both DPP and Prawitz, he generalized ground resolution so that instead of clauses having to be grounded to use resolution, resolution was lifted directly to the Skolem form [Rob63, Rob65] (1963,1965). This is, of course, a landmark result in mechanically proving first-order logic sentences.

Resolution was extended by Tseitin in [Tse68] (1968) who showed that, for any pair of variables a, b in a given CNF expression ϕ , the following expression may be appended to ϕ :

$$(z \vee a) \wedge (z \vee b) \wedge (\neg z \vee \neg a \vee \neg b)$$

where z is a variable not in ϕ . The meaning of this expression is: either a and b both have value 1 or at least one of a or b has value 0. It may be written $z \Leftrightarrow \neg a \vee \neg b$. It is safe to append such an expression because its three clauses can always be forced to value 1 by setting free variable z to the value of $\neg a \vee \neg b$. More generally, any expression of the form

$$z \Leftrightarrow f(a, b, \dots)$$

may be appended, where f is some arbitrary Boolean function and z is a new, free variable. Judicious use of such extensions can result in polynomial size refutations for problems that have no polynomial size refutations without extension. A

notable example is the pigeon hole formulas. Tseitin also showed that by adding variables not in ϕ , one can obtain, in linear time, a satisfiability-preserving translation from any propositional expression to CNF with at most a constant factor blowup in expression size.

After this point the term satisfiability was used primarily to describe the problem of finding a model for a Boolean expression. The complexity of Satisfiability became the major issue due to potential practical applications for Satisfiability. Consequently, work branched in many directions. The following sections describe most of the important branches.

1.16. The complexity of resolution

by Alasdair Urquhart

Perhaps the theoretically deepest branch is the study of the complexity of resolution. As seen in previous sections, the question of decidability dominated research in logic until it became important to implement proof systems on a computer. Then it became clear empirically and theoretically through the amazing result of Cook [Coo71] (1971) that decidable problems could still be effectively unsolvable due to space and time limitations of available machinery. Thus, many researchers turned their attention to the complexity issues associated with implementing various logic systems. The most important of these, the most relevant to the readers of this chapter, and the subject of this section is the study of resolution refutations of contradictory sets of CNF clauses (in this section clause will mean CNF clause).

Rephrasing the “rule for eliminating atomic formulas” from the previous section: if $A \vee l$ and $B \vee \neg l$ are clauses, then the clause $A \vee B$ may be inferred by the resolution rule, *resolving on* the literal l . A *resolution refutation* of a set of clauses Σ is a proof of the empty clause from Σ by repeated applications of the resolution rule.

Refutations can be represented as trees or as sequences of clauses; the worst case complexity differs considerably depending on the representation. We shall distinguish between the two by describing the first system as “tree resolution,” the second simply as “resolution.”

Lower bounds on the size of resolution refutations provide lower bounds on the running time of algorithms for the Satisfiability problem. For example, consider the familiar DPLL algorithm that is the basis of many of the most successful algorithms for Satisfiability. If a program based on the splitting rule terminates with the output “The set of clauses Σ is unsatisfiable,” then a trace of the program’s execution can be given in the form of a binary tree, where each of the nodes in the tree is labeled with an assignment to the variables in Σ . The root of the tree is labeled with the empty assignment, and if a node other than a leaf is labeled with an assignment ϕ , then its children are labeled with the assignments $\phi[v := 0]$ and $\phi[v := 1]$ that extend ϕ to a new variable v ; the assignments labeling the leaves all falsify a clause in Σ . Let us call such a structure a “semantic tree,” an idea introduced by Robinson [Rob68] and Kowalski and Hayes [KH69].

A semantic tree for a set of clauses Σ can be converted into a tree resolution refutation of Σ by labeling the leaves with clauses falsified by the assignment at

the leaves, and then performing resolution steps corresponding to the splitting moves (some pruning may be necessary in the case that a literal is missing from one of the premisses). It follows that a lower bound on the size of tree resolution refutations for a set of clauses provides a lower bound on the time required for a DPLL-style algorithm to certify unsatisfiability. This lower bound applies no matter what strategies are employed for the order of variable elimination.

The first results on the complexity of resolution were proved by Grigori Tseitin in 1968 [Tse68]. In a remarkable pioneering paper, Tseitin showed that for all $n > 0$, there are contradictory sets of clauses Σ_n , containing $O(n^2)$ clauses with at most four literals in each clause, so that the smallest tree resolution refutation of Σ_n has $2^{\Omega(n)}$ leaves. Tseitin's examples are based on graphs. If we assign the values 0 and 1 to the edges of a finite graph G , we can define a vertex v in the graph to be *odd* if there are an odd number of vertices attached to v with the value 1. Then Tseitin's clauses $\Sigma(G)$ can be interpreted as asserting that there is a way of assigning values to the edges so that there are an odd number of odd vertices. The set of clauses Σ_n mentioned above is $\Sigma(G_n)$, where G_n is the $n \times n$ square grid.

Tseitin also proved some lower bounds for resolution but only under the restriction that the refutation is *regular*. A resolution proof contains an *irregularity* if there is a sequence of clauses C_1, \dots, C_k in it, so that C_{i+1} is the conclusion of a resolution inference of which C_i is one of the premisses, and there is a variable v so that C_1 and C_k both contain v , but v does not occur in some intermediate clause C_j , $1 < j < k$. In other words, an irregularity occurs if a variable is removed by resolution, but is later introduced again in a clause depending on the conclusion of the earlier step. A proof is *regular* if it contains no irregularity. Tseitin showed that the lower bound for Σ_n also applies to regular resolution. In addition, he showed that there is a sequence of clauses Π_n so that there is a superpolynomial speedup of regular resolution over tree resolution (that is to say, the size of the smallest tree resolution refutation of Π_n is not bounded by any fixed power of the size of the smallest regular refutation of Π_n).

Tseitin's lower bounds for the graph-based formulas were improved by Zvi Galil [Gal77], who proved a truly exponential lower bound for regular resolution refutations of sets of clauses based on expander graphs E_n of bounded degree. The set of clauses $\Sigma(E_n)$ has size $O(n)$, but the smallest regular resolution refutation of $\Sigma(E_n)$ contains $2^{\Omega(n)}$ clauses.

The most important breakthrough in the complexity of resolution was made by Armin Haken [Hak85], who proved exponential lower bounds for the pigeonhole clauses PHC_n . These clauses assert that there is an injective mapping from the set $\{1, \dots, n+1\}$ into the set $\{1, \dots, n\}$. They contain $n+1$ clauses containing n literals asserting that every element in the first set is mapped to some element of the second, and $O(n^3)$ two-literal clauses asserting that no two elements are mapped to the same element of $\{1, \dots, n\}$. Haken showed that any resolution refutation of PHC_n contains $2^{\Omega(n)}$ clauses.

Subsequently, Urquhart [Urq87] adapted Haken's argument to prove a truly exponential lower bound for clauses based on expander graphs very similar to those used earlier by Galil. The technique used in Urquhart's lower bounds were employed by Chvátal and Szemerédi [CS88] to prove an exponential lower bound

on random sets of clauses. The model of random clause sets is that of the constant width distribution discussed below in Section 1.18. Their main result is as follows: if c, k are positive integers with $k \geq 3$ and $c2^{-k} \geq 0.7$, then there is an $\epsilon > 0$, so that with probability tending to one as n tends to infinity, the random family of cn clauses of size k over n variables is unsatisfiable and its resolution complexity is at least $(1 + \epsilon)^n$.

The lower bound arguments used by Tseitin, Galil, Haken and Urquhart have a notable common feature. They all prove lower bounds on size by proving lower bounds on width – the *width* of a clause is the number of literals it contains, while the *width* of a set of clauses is the width of the widest clause in it. If Σ is a contradictory set of clauses, let us write $w(\Sigma)$ for the width of Σ , and $w(\Sigma \vdash 0)$ for the minimum width of a refutation of Σ .

The lower bound techniques used in earlier work on the complexity of resolution were generalized and unified in a striking result due to Ben-Sasson and Wigderson [BSW01]. If Σ is a contradictory set of clauses, containing the variables V , let us write $S(\Sigma)$ for the minimum size of a resolution refutation of Σ . Then the main result of [BSW01] is the following lower bound:

$$S(\Sigma) = \exp \left(\Omega \left(\frac{(w(\Sigma \vdash 0) - w(\Sigma))^2}{|V|} \right) \right).$$

This lower bound easily yields the lower bounds of Urquhart [Urq87], as well as that of Chvátal and Szemerédi [CS88] via a width lower bound on resolution refutations.

Tseitin was the first to show a separation between tree resolution and general resolution, as mentioned above. The separation he proved is fairly weak, though superpolynomial. Ben-Sasson, Impagliazzo and Wigderson [BSIW04] improved this to a truly exponential separation between the two proof systems, using contradictory formulas based on pebbling problems in directed acyclic graphs.

These results emphasize the inefficiency of tree resolution, as opposed to general resolution. A tree resolution may contain a lot of redundancy, in the sense that the same clause may have to be proved multiple times. The same kind of inefficiency is also reflected in SAT solvers based on the DPLL framework, since information accumulated along certain branches is immediately discarded. This observation has led some researchers to propose improved versions of the DPLL algorithm, in which such information is stored in the form of clauses. These algorithms, which go under the name “clause learning,” lead to dramatic speedups in some cases – the reader is referred to the paper of Beame, Kautz and Sabharwal [BKS03] for the basic references and some theoretical results on the method.

1.17. Refinement of Resolution-Based SAT Solvers

Most of the current interest in Satisfiability formulations and methods is due to refinements to the basic DPLL framework that have resulted in speed-ups of many orders of magnitude that have turned many problems that were considered intractable in the 1980s into trivially solved problems now. These refinements

include improvements to variable choice heuristics, early pruning of the search space, and replacement of a tree-like search space with a DAG-like search space.

DPLL (1962) was originally designed with two important choice heuristics: the pure-literal rule and the unit-clause rule (Both described on Page 20). But, variable choices in case no unit clauses or pure literals exist were undefined. Thus, extensions of both heuristics were proposed and analyzed in the 1980s through the early 2000s. For example, the unit-clause rule was generalized to the *shortest-clause rule*: choose a variable from an existing clause containing the fewest unset literals [CF90, CR92] (1990, described on Page 36) and the pure-literal rule was extended to *linear autarkies* [vM00] (2000, described on Page 29). Other notable analyzed heuristics include the *majority rule*: choose a variable with the maximum difference between the number of its positive and negative literals [CF86] (1986, see also Page 37), probe-order backtracking [PH97] (1997, see also Page 36), and a greedy heuristic [HS03, KKL04] (2003, described on Page 38). The above heuristics represent parts of many heuristics that have actually been implemented in SAT solvers; they were studied primarily because their performance could be analyzed probabilistically.

Early heuristics designed to empirically speed up SAT solvers were based on the idea that eliminating small clauses first tends to reveal inferences sooner. Possibly the earliest well-known heuristic built on this approach is due to Jeroslow and Wang [JW90] (1990): they choose a value for a variable that comes close to maximizing the chance of satisfying the remaining clauses, assuming the remaining clauses are statistically independent in the sense described on Page 37. They assign weights $w(\mathcal{S}_{i,j})$ for each variable v_j and each value $i \in \{0, 1\}$ where, for a subset of clauses \mathcal{S} , $\mathcal{S}_{i,j}$ is the clauses of \mathcal{S} less those clauses satisfied and those literals falsified by assigning value i to v_j and $w(\mathcal{S}_{i,j}) = \sum_{C \in \mathcal{S}_{i,j}} 2^{-|C|}$ ($|C|$ is the width of clause C). The Jeroslow-Wang heuristic was intended to work better on satisfiable formulas but Böhm [BKB92] (1992) and Freeman [Fre95] (1995) came up with heuristics that work better on unsatisfiable formulas: they choose to eliminate a variable that in some sense maximally causes an estimate of both sides of the resulting search space to be roughly of equal size. This can be done, for example, by assigning variable weights that are the product of positive and negative literal weights as above and choosing the variable of maximum weight. Later, it was observed that the activity of variable assignment was an important factor in search space size. This led to the DLIS heuristic of GRASP [MSS96] and eventually to the VSIDS heuristic of Chaff [MMZ⁺01]. Most recent DPLL based SAT solvers use variations of the VSIDS branching heuristic.

Another important refinement to SAT solvers is the early generation of inferences during search. This has largely taken the appearance of lookahead techniques. It was Stålmärck [SS98] who demonstrated the effectiveness of lookahead in Prover (1992): the particular type of lookahead used there is best described as breadth-first. This scheme is limited by the fact that search space width can be quite large so the lookahead must be restricted to a small number of levels, usually 2. Nevertheless, Prover was regarded to be a major advance at the time. Later, Chaff used depth-first lookahead (this form is generally called restarts) to greater effectiveness [MMZ⁺01]. Depth-first lookahead also solves a problem that comes up when saving non-unit inferences (see below): such inferences may

be saved in a cache which may possibly overflow at some point. In the case of depth-first lookahead, the inferences may be applied and search restarted with a cleared cache.

Extremely important to SAT solver efficiency are mechanisms that reduce the size of the search space: that is, discover early that branch of the search space does not have to be explored. A *conflict analysis* at a falsified node of the search space will reveal the subset V_s of variables involved in that node becoming false. Thus, the search path from root to that falsified node can be collapsed to include only the variables in V_s , effectively pruning many branch points from the search space. This has been called non-chronological backtracking [MSS96] (1996) and is seen in all of the most efficient DPLL based SAT solvers. The result of conflict analysis can also be saved as a non-unit inference in the form of a clause. If later in the search a subset of variables are set to match a saved inference, backtracking can immediately be applied to avoid repeating a search that previously ended with no solution. This idea, known as *clause learning*, is an essential part of all modern DPLL based SAT solvers: its power comes from turning what would be a tree-like search space into a DAG-like search space. This has gone a long way toward mitigating the early advantage of BDDs (see Section 1.20) over search: BDD structures are non-repetitive. In practice, clause learning is often implemented jointly with the opportunistic deletion of the less used learnt clauses, thus reducing the over-usage of physical memory by SAT solvers. Nevertheless, several of the most efficient DPLL based SAT solvers opt for not deleting any of the learnt clauses, and so the need for deleting less used learnt clauses is still not a completely solved issue.

Finally, the development of clever data structures have been crucial to SAT solver success. The most important of these is the structure that supports the notion of the watched literal [MMZ⁺01].

1.18. Upper bounds

by Ewald Speckenmeyer

Deciding satisfiability of a CNF formula ϕ with n Boolean variables can be performed in time $O(2^n |\phi|)$ by enumerating all assignments of the variables. The number m of clauses as well as the number l of literals of ϕ are further parameters for bounding the runtime of decision algorithms for SAT. Note that l is equal to $\text{length}(\phi)$ and this is the usual parameter for the analysis of algorithms. Most effort in designing and analyzing algorithms for SAT solving, however, are based on the number n of variables of ϕ .

A first non-trivial upper bound of $O(\alpha_k^n \cdot |\phi|)$, where α_k^n is bounding the Fibonacci-like recursion

$$\begin{aligned} T(1) &= T(2) = \dots = T(k-1) = 1 \quad \text{and} \\ T(n) &= T(n-1) + T(n-2) + \dots + T(n-k+1), \quad \text{for } n \geq k, \end{aligned}$$

for solving k -SAT, $k \geq 3$, was shown in [MS79, MS85]. For example, $\alpha_3 \geq 1.681$, $\alpha_4 \geq 1.8393$, and $\alpha_5 \geq 1.9276$.

The algorithm supplying the bound looks for a shortest clause c from the current formula. If c has $k - 1$ or less literals, e.g. $c = (x_1 \vee \dots \vee x_{k-1})$ then ϕ is split into $k - 1$ subformulas according to the $k - 1$ subassignments

- 1) $x_1 = 1$;
- 2) $x_1 = 0, x_2 = 1$;
- ...
- $k-1$) $x_1 = x_2 = \dots = x_{k-2} = 0, x_{k-1} = 1$.

If all clauses c of ϕ have length k , then ϕ is split into k subformulas as described, and each subformula either contains a clause of length at most $k - 1$ or one of the resulting subformulas only contains clauses of length k . In the former case, the above mentioned bound holds. In the latter case, the corresponding subassignment is *autark*, that is, all clauses containing these variables are satisfied by this subassignment, so ϕ can be evaluated according to this autark subassignment thereby yielding the indicated upper bound.

For the case $k = 3$, the bound was later improved to $\alpha_3 = 1.497$ by a sophisticated case analysis (see [Sch96]). Currently, from [DGH⁺02], the best deterministic algorithms for solving k -SAT have a run time of

$$O\left(\left(2 - \frac{2}{k+1}\right)^n\right).$$

For example, the bounds are $O(1.5^n)$, $O(1.6^n)$, $O(1.666\dots^n)$ for 3, 4, and 5 literal clause formulas. These bounds were obtained by the derandomization of a multistart-random-walk algorithm based on covering codes. In the case of $k = 3$ the bound has been further improved to $O(1.473^n)$ in [BK04]. This is currently the best bound for deterministic 3-SAT solvers.

Two different probabilistic approaches to solving k -SAT formulas have resulted in better bounds and paved the way for improved deterministic bounds. The first one is the algorithm of Paturi-Pudlak-Zane [PPZ97] which is based on the following procedure:

Determine a truth assignment of the variables of the input formula ϕ by iterating over all variables v of ϕ in a randomly chosen order: If ϕ contains a unit clause $c = (x)$, where $x = v$ or $\neg v$, set $t(v)$ such that $t(x) = 1$. If neither v nor $\neg v$ occur in a unit clause of ϕ , then randomly choose $t(v)$ from $\{0, 1\}$. Evaluate $\phi := t(\phi)$. Iterate this assignment procedure at most r times or until a satisfying truth assignment t of ϕ is found, starting each time with a randomly chosen order of variables for the assignment procedure.

After $r = 2^{n(1-\frac{1}{k})}$ rounds a solution for a satisfiable formula is found with high probability [PPZ97]. By adding to ϕ clauses originating from resolving input clauses up to a certain length this bound can be improved. In case of $k = 3, 4, \text{ and } 5$, the basis of the exponential growth function is 1.36406, 1,49579, and 1,56943 [PPZ98]. For $k \geq 4$ this is currently the best probabilistic algorithm for solving k -SAT.

The second approach is due to Schöning [Sch99, Sch02] and extends an idea

of Papadimitriou [Pap91]. That procedure is outlined as follows:

Repeat the following for r rounds: randomly choose an initial truth assignment t of ϕ ; if t does not satisfy ϕ , then repeat the following for three times the number of variables of ϕ or until a satisfying assignment t is encountered: select a falsified clause c from ϕ and randomly choose and flip a literal x of c .

If the algorithm continues, round after round, without finding a satisfying assignment nothing definite can be said about the satisfiability of ϕ . However, in this case the higher the number of rounds r , the lower the probability that ϕ is satisfiable. To guarantee an error probability of $e^{-\lambda}$, the number of rounds should be at least $O(\lambda(2^{\frac{k-1}{k}})^n)$. For $k = 3, 4,$ and 5 the basis of the exponential growth is 1.3334, 1.5, and 1.6.

The case of $k = 3$ has since been improved to $O(1.324^n)$ by Iwama and Tamaki [IT03]. For CNF formulas of unrestricted clause length the best time bound for a deterministic solution algorithm is $O(2^{n(1-\frac{1}{\log(2m)})})$, where n is the number of variables and m the number of clauses. This result, by Dantsin and Wolpert [DW04], is obtained from a derandomization of a randomized algorithm by Schuler [Sch05], of the same time complexity. Dantsin and Wolpert recently improved this bound for randomized algorithms to $O\left(2^{n(1-\frac{1}{\ln(\frac{m}{n})+O(\frac{1}{\ln(\ln(m))})})}\right)$ [DW05].

1.19. Classes of easy expressions

An entire book could be written about the multitude of classes of the Satisfiability problem that can be solved in polynomial time, so only some of the classical results will be mentioned here. It is often not enough that a class of problems is solved in polynomial time - an instance may have to be recognized as a member of that class before applying an efficient algorithm. Perhaps surprisingly, for some classes the recognition step is unnecessary and for some it is necessary but not known to be tractable.

The reader may have the impression that the number of polynomial time solvable classes is quite small due to the famous dichotomy theorem of Schaefer [Sch73]. But this is not the case. Schaefer proposed a scheme for defining classes of propositional expressions with a generalized notion of “clause.” He proved that every class definable within his scheme was either \mathcal{NP} -complete or polynomial-time solvable, and he gave criteria to determine which. But not all classes can be defined within his scheme. The Horn and XOR classes can be but we will describe several others including q-Horn, extended Horn, CC-balanced and SLUR that cannot be so defined. The reason is that Schaefer’s scheme is limited to classes that can be recognized in log space.

All clauses of a 2-SAT expression contain at most two literals. A two literal clause describes two inferences. For example, inferences for the clause $(x \vee y)$ are: 1) if x is 0 then y is 1; and 2) if y is 0 then x is 1. For a given 2-SAT expression an implication graph may be constructed where directed edges between pairs of literals represent all the inferences of the expression. A cycle in the inference

graph that includes a pair of complementary literals is proof that no model exists. Otherwise a model may be determined easily with a depth-first search of the graph, which amounts to applying unit propagation. A full algorithm is given in [EIS76] (1976). A linear time algorithm is given in [APT79] (1979).

All clauses of an expression said to be *Horn* have at most one *positive literal*. This class is important because of its close association with Logic Programming: for example, the clause $(\neg v_1 \vee \neg v_2 \vee v)$ expresses the rule $v_1 \wedge v_2 \rightarrow v$ or $v_1 \rightarrow v_2 \rightarrow v$. However, the notion of causality is generally lost when translating from rules to Horn expressions. An extremely important property of Horn expressions is that every satisfiable one has a unique minimum model with respect to 1 (the unique minimum model is the intersection of all models of the expression). Finding a model is a matter of applying unit propagation on positive literals until all positive literals are eliminated, then assigning all remaining literals the value 0 (if satisfiable, a unique minimum model is the result). It took a few iterations in the literature to get this universally understood and the following are the important citations relating to this: [IM82] (1982), [DG84] (1984), [Scu90] (1990).

A given expression may be *renameable Horn*, meaning a change in the polarity of some variables results in an equivalent Horn expression. Renameable Horn expressions were shown to be recognized and solved in linear time by [Lew78] (1978) and [Asp80] (1980).

A number of polynomially solvable relaxations of Linear Programming problems were shown to be equivalent to classes of Satisfiability; this work, quite naturally, originated from the Operations Research community. Representing CNF expressions as $(0, \pm 1)$ matrices where columns are indexed on variables and rows indexed on clauses, Satisfiability may be cast as an Integer Programming problem. If the matrix has a particular structure the Integer Program can be relaxed to a Linear Program, solved, and the non-integer values of the solution rounded to 0 or 1. Notable classes based on particular matrix structures are the extended Horn expressions and what we call the CC-balanced expressions.

The class of *extended Horn* expressions was introduced by Chandru and Hooker [CH91] (1991). Their algorithm is due to a theorem of Chandrasekaran [Cha84] (1984). Essentially, a model for a satisfiable expression may be found by applying unit propagation, setting values of unassigned variables to $1/2$ when no unit clauses remain, and rounding the result by a matrix multiplication. This algorithm cannot, however, be reliably applied unless it is known that a given expression is extended Horn and, unfortunately, the problem of recognizing an expression as extended Horn is not known to be solved in polynomial time.

The class of *CC-balanced* expressions has been studied by several researchers (see [CCK⁺94] (1994) for a detailed account of balanced matrices and a description of CC-balanced formulas). The motivation for this class is the question, for Satisfiability, when do Linear Programming relaxations have integer solutions? The satisfiability of a CNF expression can be determined in linear time if it is known to be CC-balanced and recognizing that a formula is CC-balanced takes linear time.

Horn, Renameable Horn, Extended Horn, CC-balanced expressions, and other classes including that of [SW95] (1991) turn out to be subsumed by a larger, efficiently solved class called *SLUR* for Single Lookahead Unit Resolution [SAFS95]

(1995). The SLUR class is peculiar in that it is defined based on an algorithm rather than on properties of expressions. The SLUR algorithm recursively selects variables sequentially and arbitrarily, and considers a one-level lookahead, under unit propagation in both directions, choosing only one, if possible. If a model is found, the algorithm is successful, otherwise it “gives up.” An expression is SLUR if, for all possible sequences of variable selections, the SLUR algorithm does not give up. Observe that due to the definition of this class, the question of class recognition is avoided. In fact, SLUR provides a way to avoid preprocessing or recognition testing for several polynomial time solvable classes of SAT when using a reasonable variant of the DPLL algorithm.

The worst case time complexity of the SLUR algorithm would appear to be quadratic. However, a simple modification brings the complexity down to linear: run both calls of unit propagation simultaneously, alternating execution of their repeat blocks. When one terminates without an empty clause in its output formula, abandon the other call.

The *q-Horn* class also originated in the Operations Research community [BCH90, BHS94] (1990) and for several years was thought to be what was described as the largest, succinctly expressed class of polynomial time solvable expressions. This claim was due to a measure on an underlying $(0, \pm 1)$ matrix representation of clauses called the *satisfiability index* [BCHS94] (1994). The q-Horn class was also studied as a special case of the maximum monotone decomposition of matrices [Tru94] (1994). We find it easier to describe the efficient solution of q-Horn expressions by following [Tru98] (1998). Using the $(0, \pm 1)$ matrix representation, an expression is q-Horn if columns can be multiplied by -1 , and permuted, and rows can be permuted with the result that the matrix has four quadrants as follows: northeast - all 0s; northwest - a Horn expression; southeast - a 2-SAT expression; southwest - no $+1$ s. A model may be found in linear time, if one exists, by finding a model for the northwest quadrant Horn expression, cancelling rows in the southern quadrants whose clauses are satisfied by that model, and finding a model for the southeast quadrant 2-SAT expression. It was shown in [BCHS94] (1994) that a CNF expression is q-Horn if and only if its satisfiability index is no greater than 1 and the class of all expressions with a satisfiability index greater than $1 + 1/n^\epsilon$, for any fixed $\epsilon < 1$, is \mathcal{NP} -complete. The SLUR and q-Horn classes are incomparable [FVG03] (2003).

The class of *linear autarkies*, developed by Kullmann [Kul00] (2000), was shown by van Maaren [vM00] (2000) to include the class of q-Horn formulas. It was also shown to be incomparable with the SLUR class. An autarky for a CNF formula ϕ is a partial assignment that satisfies all those clauses of ϕ affected by it: for example, a pure literal is an autarky. Therefore, a subformula obtained by applying an autarky to ϕ is satisfiable if and only if ϕ is. A formula with $(0, \pm 1)$ matrix representation A has a linear autarky $x \in Q^n$, $x \neq 0$, if $Ax \geq 0$. in polynomial time. There exists a simple, efficient decomposition that results in a partial, autark assignment. Applying this decomposition repeatedly results in a unique, linear-autarky-free formula. If the decomposition is repeatedly applied to a renameable Horn formula without unit clauses what is left is empty and if it is applied repeatedly to a 2-SAT formula, the formula is unsatisfiable if what is left is not empty and satisfiable otherwise.

The class of *matched expressions* was analyzed to provide a benchmark for testing the claim made for the q-Horn class. This is a class first described in [Tov84] (1984) but not extensively studied, probably because it seems to be a rather useless and small class of formulas. Establish a bipartite graph $G_\phi = (V_1, V_2, E)$ for an expression ϕ where V_1 vertices represent clauses, V_2 vertices represent variables, and edge $\langle c, v \rangle \in E$ if and only if clause c contains v or its complement. If there is a total matching in G_ϕ , then ϕ is said to be matched. Clearly, matched expressions are always satisfiable and are trivially solved. The matched class is incomparable with the q-Horn and SLUR classes. However, as is shown in Section 1.22, with respect to frequency of occurrence on random expressions, matched expressions are far more common than both those classes together [FVG03] (2003).

The worst case complexity of *nested satisfiability*, a class inspired by Lichtenstein's theorem of planar satisfiability [Lic82] (1982), has been studied in [Knu90] (1990). Index all variables in an expression consecutively from 1 to n and let positive and negative literals take the index of their respective variables. A clause c_i is said to *straddle* another clause c_j if the index of a literal of c_j is strictly between two indices of literals of c_i . Two clauses are said to *overlap* if they straddle each other. A formula is said to be *nested* if no two clauses overlap. For example, the following formula is nested

$$(v_6 \vee \neg v_7 \vee v_8) \wedge (v_2 \vee v_4) \wedge (\neg v_6 \vee \neg v_9) \wedge (v_1 \vee \neg v_5 \vee v_{10}).$$

The class of nested formulas is quite limited in size for at least the reason that a nested expression can contain no more than $2m + n$ literals. Thus, no expression consisting of k -literal clauses is a nested formula unless $m/n < 1/(k-2)$. The class of nested expressions is incomparable with both the SLUR and q-Horn classes. However, by the measure of Section 1.22 (also in [FVG03] (2003)) a random expression is far more likely to be matched, q-Horn, or even SLUR than nested. The algorithm for nested expressions is notable for being quite different than those mentioned above: instead of relying on unit propagation, it uses dynamic programming to find a model in linear time. The question of whether the variable indices of a given formula can, in linear time, be permuted to make the formula nested appears to be open. An extension to nested satisfiability, also solvable in linear time, has been proposed in [HJP93] (1993).

None of the classes above covers a significant proportion of unsatisfiable expressions. Nevertheless, several classes of unsatisfiable expressions have been identified. It is interesting that most known polynomial time solvable classes with clauses containing three or more literals either are strongly biased toward satisfiable expressions or strongly biased toward unsatisfiable expressions.

An expression is said to be *minimally unsatisfiable* if it is unsatisfiable and removing any clause results in a satisfiable expression. A minimally unsatisfiable expression with n variables must have at least $n + 1$ clauses [AL86] (1986). Every variable of a minimally unsatisfiable expression occurs positively *and* negatively in the expression. The class of minimally unsatisfiable formulas is solved in $n^{O(k)}$ if the number of clauses exceeds the number of variables by a fixed positive constant k [KB99] (1999) and [KB00] (2000). Szeider improved this to $O(2^k)n^4$ [Sze03] (2003). Kullmann has generalized this class in [Kul03] (2003) and continues to

find larger versions. Some SAT solvers look for minimally unsatisfiable sets of clauses to reduce search space size.

A CNF expression is said to be k -BRLR if all resolvents derived from it have a number of literals bounded by k or if the null clause is derivable from resolvents having at most k literals. Obviously, this class is solved in time bounded by $2^k \binom{n}{k}$.

Finally, we mention that semidefinite programming which is discussed at length in Section 1.25 is biased toward verifying unsatisfiability and this accounts for the success of this approach.

1.20. Binary Decision Diagrams

Binary Decision Diagrams (BDDs) were considered for a while to be the best way to handle some problems that are rooted in real applications, particularly related to circuit design, testing, and verification. They are still quite useful in various roles [DK03, FKS⁺04, HD04, JS05, MM02, PV05, CGGT97] and in some ways are complementary to search [US94, GZ03].

A BDD may be regarded as a DAG representation of the truth table of a Boolean function. In a BDD non-leaf vertices are labeled as variables, there are two out-directed edges for each non-leaf vertex, each labeled with value 0 or 1, and two leaves, labeled 1 and 0. There is a single root and any path from root to a “1” (“0”) leaf indicates an assignment of values to the variables which causes the represented function to have value 1 (0). A BDD may also be viewed as representing a search space in which paths to the 1 leaf represent models.

The attraction to BDDs is due in part to the fact that no subproblem is represented more than once in a collection of BDDs - this is in contrast to the tree-like search spaces of DPLL implementations of the 1980s. In addition, efficient implementations exist for BDD operations such as existential quantification, “or,” “and,” and others. The down side is: 1) each path from the root of a BDD must obey the same variable ordering, so BDDs are not necessarily a minimal representation for a function; and 2) repeatedly conjoining pairs of BDDs may create outrageously large intermediate BDDs even though the final BDD is small. By the late 1990s, DPLL advances such as conflict resolution, clausal learning, backjumping, restarts, and more gave rise to DAG search spaces with dynamic variable ordering. The result was improved performance for search over BDDs in some cases.

BDDs were introduced in [Lee59] (1959) as a data structure based on the Shannon expansion (see Section 1.14). They were publicized in [Bou76] (1976) and [Ake78] (1978). But BDDs became most useful with the introduction of reduced order BDDs by Bryant [Bry86, Bry92] (1986,1992) and their implementation [BRB90] (1990) which supports subproblem sharing over collections of BDDs and the efficient manipulation of those BDDs.

A number of operations on BDDs have been proposed over the years to assist in the efficiency of combining BDDs. One of the most important and basic operations is **existential quantification** which arises directly from Shannon’s expansion of Section 1.14: namely, replace f with $f|_{v=0} \vee f|_{v=1}$. This operation can be used to eliminate a variable at the expense of first conjoining all BDDs containing it. It is not clear when existential quantification was first used but it

was probably known very early. An important problem in managing BDD size is how to simplify a function f (implemented as a BDD), given a constraint c (implemented as a BDD): that is, replace f with a function f' that is equal to f on the domain defined by c . The operation of **restrict** [CBM90] (1990) does this by pruning paths from the BDD of f that are ‘false’ in the BDD of c . A more complex operation with restrict-like properties but admitting removal of a BDD from a collection of BDDs at the expense of increasing the size of some of the remaining BDDs is the **constrain** or **generalized cofactor** operation [CM90] (1990). Constraining f to c results in a function h with the property that $h(x) = f(\mu(x))$ where $\mu(x)$ is the closest point to x in c where distance between binary vectors $x, y \in \{0, 1\}^n$ is measured by $d(x, y) = \sum_{1 \leq i \leq n} 2^{n-i} \cdot ((x_i + y_i) \bmod 2)$ under a BDD variable ordering which matches the index order. Additional minimization operations, based on **restrict**, that do not increase BDD size are proposed in [HBBM97] (1997).

BDDs were designed primarily for the efficient representation of switching circuits and their manipulation, but a number of variations on BDDs have appeared over the years to target related classes of problems. BDDs of various kinds have been used successfully to represent relations and formulas to support symbolic model checking [BCM⁺92, McM93] (1992), although more recently, it has been found that SAT solvers for Bounded Model Checking [BCCZ99] (1999) can sometimes achieve even better results. The ZDD, for Zero-suppressed BDD, introduced in 1993 [Min93], differs from the BDD in that a vertex is removed if its 1 edge points to the 0 leaf. This helps improve efficiency when handling sparse sets and representing covers. Thus, the ZDD has been used successfully on problems in logic synthesis such as representing an irredundant DNF of an incompletely specified Boolean function [CMFT93] (1993), finding all essential prime implicants in DNF minimization [Cou94] (1994), factorization [Min96] (1996), and decomposition [JM01] (2001). The BMD, for Binary Moment Diagram, is a generalization of the BDD for handling numbers and numeric computations, particularly multiplication [BC95] (1995). The ADD, for Algebraic Decision Diagram (also known as Multi-Terminal Decision Diagram), allows more than two leaves (possibly real-valued) and is useful for mapping Boolean functions to sets [BFG⁺93] (1993) as might be needed to model the delay of MOS circuits, for example [MB01] (2001). The XDD has been used to efficiently represent equations involving the xor operation. The essential idea is that vertex x with 1 edge to vertex a and 0 edge to vertex b represents the equation $(x \wedge a) \oplus b = 1$. Many other varieties of BDD have been proposed but there is not enough space to mention them all.

1.21. Probabilistic analysis: SAT algorithms

Probabilistic and average-case analysis of algorithms can give useful insight into the question of what SAT algorithms might be effective and why. Under certain circumstances, one or more structural properties shared by each of a collection of formulas may be exploited to solve such formulas efficiently; or structural properties might force a class of algorithms to require superpolynomial time. Such properties may be identified and then, using probabilistic analysis, one may argue that these properties are so common that the performance of an algorithm

or class of algorithms can be predicted for most of a family of formulas. The first probabilistic results for SAT had this aim.

Probabilistic results of this sort depend on an underlying input distribution. Although many have been proposed, well-developed histories exist for just two, plus a few variants, which are defined over CNF inputs. All the results of this section are based on these two. In what follows, the *width* of a clause is the number of literals it contains. The distributions we discuss here are:

1. *Variable width distribution:* Given integers n , m , and a function $p : N^+ \times N^+ \rightarrow [0, 1]$. Let $V = \{v_1, \dots, v_n\}$ be a set of n Boolean variables. Construct a random clause (disjunction) c by independently adding literals to c as follows: for each variable v_i , $1 \leq i \leq n$, add literal v_i with probability $p(m, n)$, add literal $\neg v_i$ with probability $p(m, n)$ (therefore add neither v_i nor $\neg v_i$ with probability $1 - 2p(m, n)$). A random input consists of m independently constructed random clauses and is referred to below as a random (n, m, p) -CNF expression. In some cases $p(m, n)$ is independent of m and n and then we use p to represent $p(m, n)$.
2. *Constant width distribution:* Given integers n , m , k . Let $V = \{v_1, \dots, v_n\}$ be a set of n Boolean variables. Construct a random clause (disjunction) c by choosing, uniformly and without replacement, k variables from V and then complementing each, independently, with probability $1/2$. A random input consists of m independently constructed random clauses and is referred to below as a random (n, m, k) -CNF expression. Such an input is widely known as a uniform random k -SAT instance or simply random k -SAT instance.

1.21.1. Variable width distribution

Goldberg was among the first to apply probability to the analysis of SAT algorithms. He investigated the frequency with which simple backtracking returns a model quickly on random CNF formulas by providing an average-case analysis of a variant of DPLL which does not handle pure literals or unit clauses [Gol79] (1979). The result received a lot of attention when it was first presented and even 10 years afterward some people still believed it was the definitive probabilistic result on satisfiability algorithms.

Goldberg showed that, for random (n, m, p) -CNF expressions, the DPLL variant has average complexity bounded from above by $O(m^{-1/\log(p)}n)$ for any fixed $0 < p < 1$. This includes the “unbiased” sample space when $p = 1/3$ and all expressions are equally likely. Later work [GPB82] (1982) showed the same average-case complexity even if pure literals are handled as well. Very many problems confronting the scientific and engineering communities are unsatisfiable, but Goldberg made no mention of the frequency of occurrence of unsatisfiable random (n, m, p) -CNF expressions.

However, Franco and Paull [FP83] (1983) pointed out that large sets of random (n, m, p) -CNF expressions, for fixed $0 < p < 1/2$, are dominated by trivial *satisfiable* expressions: that is, any random assignment of values to the variables of such a random expression is a model for that expression with high probability. This result is refined somewhat in [Fra86b] (1986) where it is shown that a

random assignment is a model for a random (n, m, p) -CNF expression with high probability if $p > \ln(m)/n$ and a random expression is unsatisfiable with high probability if $p < \ln(m)/2n$. In the latter case, a “proof” of unsatisfiability is trivially found with high probability because a random (n, m, k) -CNF expression for this range of p usually contains at least one empty clause, which can easily be located, and implies unsatisfiability. The case $p = c \ln(m)/n$, $1/2 \leq c \leq 1$ was considered in [FH89] (1988) where it was shown that a random (n, m, p) -CNF expression is satisfiable with high probability if $\lim_{n, m \rightarrow \infty} m^{1-c}/n^{1-\epsilon} < \infty$, for any $0 < \epsilon < 1$.

Although these results might be regarded as early threshold results (see Section 1.22) the main impact was to demonstrate that probabilistic analysis can be highly misleading and requires, among other things, some analysis of input distribution to ensure that a significant percentage of non-trivial inputs are generated. They show that random (n, m, p) -CNF expressions, satisfiable or unsatisfiable, are usually trivially solved because either they contain empty clauses (we get the same result even if empty or unit clauses are disallowed) or they can be satisfied by a random assignment. In other words, only a small region of the parameter space is capable of supporting significantly many non-trivial expressions: namely, when the average clause width is $c \ln(m)/n$, $1/2 \leq c \leq 1$. These results demonstrate shortcomings in choosing random (n, m, p) -CNF expressions for analysis and, because of this, such generators are no longer considered interesting by many.

Nevertheless, some interesting insights were developed by further analysis and we mention the most significant ones here. The results are shown graphically in Figure 1.1 which partitions the entire parameter space of the variable width distribution according to polynomial-average-time solvability. The vertical axis ($p \cdot n$) measures average clause width and the horizontal axis (m/n) measures density. Each result is presented as a line through the chart with a perpendicular arrow. Each line is a boundary for the algorithm labeling the line and the arrow indicates that the algorithm has polynomial average time performance in that region of the parameter space that is on the arrow side of the line (constant and even log factors are ignored for simplicity).

Goldberg’s result is shown as the diagonal line in the upper right corner of the figure and is labeled **Goldberg**: it is not even showable as a region of the parameter space, so there is no arrow there. Iwama analyzed an algorithm which counts models using inclusion-exclusion [Iwa89] (1989) and has polytime-average-time complexity in the region above and to the right of the line labeled **Counting**. A random expression generated at a point in that region satisfies conditions, with high probability, under which the number of terms in the inclusion-exclusion expansion is polynomial in m and n . However, every clause of the same random expression is a tautology (contains a literal and its complement) with high probability. Therefore, this seems to be another example of a misleading result and, judging from the relation between the **Counting** and **Goldberg** regions in the figure, lessens the significance of Goldberg’s result even more.

There have been a number of schemes proposed for limiting resolution steps to obtain polynomial complexity. A simple example is to perform resolution only if the pivot variable appears once as a positive literal and once as a negative literal - then, if an empty clause does not exist when no more resolutions are possible,

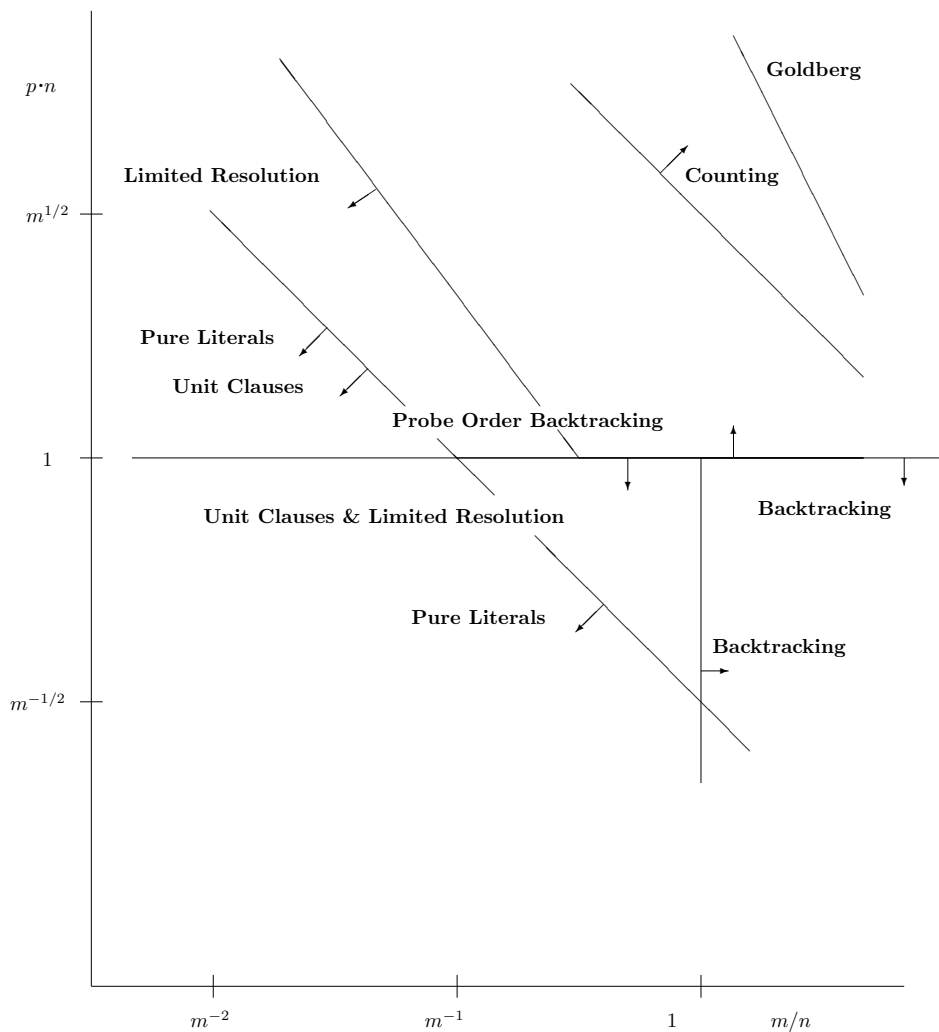


Figure 1.1. The parameter space of the variable width distribution partitioned by polynomial-average-time solvability. Pick a point in the parameter space. Locate the lines with names of algorithms on the side of the line facing the chosen point. Random formulas generated with parameters set at that point are solved in polynomial average time by the named algorithms.

do a full search for a solution. The conditions under which this algorithm runs in polynomial average time under random variable width expressions are too complex to be given here but they are represented by the regions in Figure 1.1 below the lines labeled **Limited Resolution** [Fra86a] (1991).

If pure literal reductions are added to the algorithm analyzed by Goldberg then polynomial average time is achieved in the **Pure Literals** region, considerably improving the result of Goldberg [PB85] (1985). But a better result, shown as the region bounded from above by the lines labeled **Unit Clauses**, is obtained by performing unit propagation [Fra93] (1993) and is consistent with the empirically and theoretically demonstrated importance of unit propagation in DPLL algorithms. Results for Search Rearrangement Backtracking [Pur83] (1983) are disappointing (shown bounded by the two lines labeled **Backtracking**) but a slightly different variant in which only variables from positive clauses (a *positive clause* contains only positive literals) are chosen for elimination (if there is no positive clause, satisfiability is determined by assigning all unassigned variables the value 0), has spectacular performance as is shown by the line labeled **Probe Order Backtracking** [PH97] (1997). The combination of probe order backtracking and unit clauses yield polynomial average time for random (n, m, p) -CNF expressions at every point of the parameter space except for some points along the thin horizontal region $p \cdot n = c \cdot \log(n)$.

By 1997, what Goldberg had set out to do in 1979, namely show DPLL has polynomial average time complexity for variable width distributions, had been accomplished. The research in this area had some notable successes such as the analysis of probe order backtracking which demonstrated that a slight algorithmic change can have a major effect on performance. But, the fact that nearly every point in the parameter space is covered by an algorithm that has polynomial average time complexity is rather disturbing, though, since there are many practical applications where SAT is considered extremely difficult. Largely for this reason attention shifted to the constant width distribution and, by the middle of the 1990s, research on the variable width distribution all but disappeared.

Constant width distribution

Franco and Paull (1983) in [FP83] (see [PRF87], 1987, for corrections) set out to test Goldberg's result on other distributions. They formulated the constant width distribution and showed that for all $k \geq 3$ and every fixed $m/n > 0$, with probability $1 - o(1)$, the algorithm analyzed by Goldberg takes an *exponential* number of steps to report a result: that is, either to report all ("cylinders" of) models, or that no model exists. They also showed that a random (n, m, k) -CNF expression is unsatisfiable with probability tending to 1 when $m/n > -\ln(2)/\ln(1 - 2^{-k})$. For $k = 3$ this is $m/n > 5.19$.

Perhaps the first positive result using the constant width distribution was presented in [CF86] (1986) where a non-backtracking, incremental assignment algorithm employing unit propagation, called UC, was shown to find a model with bounded probability when $m/n < O(1) \cdot 2^k/k$. This work introduced an analysis tool called "clause-flows" analysis which was refined by Achlioptas [Ach01] (2001) using a theorem of Wormald [Wor95]. Analysis of clause flows via differential equations was to be the basic analytic tool for most of the following results in this area. In [CF90] (1990) it was shown that a generalization of unit propagation,

called GUC, in which a variable from a “smallest” clause is chosen for assignment finds a model with probability $1 - o(1)$ when $m/n < O(1) \cdot 2^k/k$, $k \geq 4$. This was improved by Chvátal and Reed [CR92] (1992) who showed that GUC, with assignment rules relaxed if no unit or two literal clauses are present (called SC for shortest clause), finds a model with probability $1 - o(1)$ when $m/n < O(1) \cdot 2^k/k$ for $k \geq 3$.

Observe that the results of the previous two paragraphs show a gap between the range $m/n > \ln(2)/\ln(1 - 2^{-k}) \approx 2^k \ln(2)$, where random expressions are unsatisfiable with high probability, and $m/n < O(1) \cdot 2^k/k$, where analyzed non-backtracking incremental assignment algorithms find models for random expressions with high probability. According to results of Friedgut (1999 - see Section 1.22) there is a sharp satisfiability threshold r_k for every $k \geq 2$ (it is still not known whether the threshold depends on n) and the results above show $2^k/k < r_k < 2^k \ln(2)$. The question of the location of r_k was pondered for quite some time. Most people intuitively believed that r_k was located near $2^k \ln(2)$ so the question became known as the ‘*Why 2 k?*’ problem (Pittel, 1999). This was verified by Achlioptas, Moore, and Perez in [AM02, AP04] (2002,2004) who showed that $r_k = 2^k \ln(2) - O(k)$ by applying the second moment method to a symmetric variant of satisfiability, known as not-all-equal k -SAT (NAE- k -SAT). In NAE- k -SAT the question is whether, for a given CNF expression, there is a satisfying assignment whose complement is also a satisfying assignment. Obviously, the class of k -CNF expressions satisfiable in NAE- k -SAT is a subset of the class satisfiable in the traditional sense. Hence, for the purposes of finding the threshold, NAE- k -SAT may be analyzed instead of k -SAT. The symmetric nature of NAE- k -SAT results in a low variance in the number of satisfying assignments and this makes the second moment method work.

Most interest, though, has centered on the case $k = 3$. It is believed, from experiment, that $r_3 \approx 4.25$. In [CF86] (1986) it is shown that UC finds a model with bounded probability when $m/n < 8/3 = 2.66..$ and, when combined with a “majority” rule (that is, choose a variable with the maximum difference between the number of its positive and negative literals if unit propagation cannot be applied), this improves to $m/n < 2.9$. Frieze and Suen [FS96] (1996) considered SC and GUC and showed that for $m/n < 3.003..$, both heuristics succeed with positive probability. Moreover, they proved that a modified version of GUC, called GUCB, which uses a limited form of backtracking, succeeds almost surely for the same range of m/n . Later, Achlioptas [Ach00] (2000) analyzed SC, modified slightly to choose two variables at a time from a clause with two unfalsified literals, for an improvement to $m/n \leq 3.145$ with probability $1 - o(1)$.

Nearly all the results above apply to *myopic* algorithms. A non-backtracking, variable assignment algorithm is called *myopic* [AS00] if, under the spectral coalescence of states, the distribution of expressions corresponding to a particular coalesced state can be expressed by its spectral components alone: that is, by the *number* of clauses of width i , for all $1 \leq i \leq k$, and the *number* of assigned variables. Being myopic considerably assists an analysis by preserving statistical independence from one algorithmic step to the next. Unfortunately, in [AS00] (2000) it is shown that no myopic algorithm can have good probabilistic performance when $m/n > 3.26$. In the same paper it is shown that at least one myopic

algorithm almost surely finds a model if $m/n < 3.26$.

However, Kaporis et al. [KKL06] (2002) found a workaround to this “barrier” and analyzed a simple non-myopic greedy algorithm for Satisfiability. They control the statistical dependence problem by considering a different generator of expressions such that probabilistic results also hold for random $(n, m, 3)$ -CNF expressions. Their result is that a greedy algorithm for satisfiability almost always finds a model when $m/n < 3.42$. Similar algorithms have been shown to find a model, almost always, when $m/n < 3.52$ [HS03, KKL04] (2003). With this analysis machinery in place, better results are expected.

Although most of the results of the previous paragraph were motivated by threshold research (see Section 1.22), their analyses can influence algorithm development. It is hoped future probabilistic results will reveal new generally useful search heuristics, or at least explain the mechanics of existing search heuristics.

Another analysis should be noted for the following non-myopic incremental assignment algorithm: repeatedly assign values to pure literals to eliminate the clauses containing them. Broder, Frieze and Upfal [BFU93] (1993) proved that this algorithm almost always finds a model when $m/n \leq 1.63$ and $k = 3$. They used Martingales to offset distribution dependency problems as pure literals are uncovered. Mitzenmacher [Mit97] (1997) showed that this bound is tight.

This section concludes with a history of results on algorithms which are intended to return certificates of unsatisfiability. Most of these results, all pessimistic, are obtained for resolution. The best result, though, is obtained for an algorithm which finds lower bounds on the number of variable assignments that must be positive and the number that must be negative (via translation to two hitting set problems) and returns “unsatisfiable” if the sum is greater than the number of variables. This result is notable also because it makes use of a highly underused analysis tool in this area: eigenvalues. Recall that nearly all random constant width expressions are unsatisfiable if $m/n > 2^k \ln(2)$.

Chvátal and Szemerédi obtained the earliest result on constant width distributions, inspired by the work reported in Section 1.16, is that, with high probability, every resolution proof for a random unsatisfiable expression is exponentially long if $\lim_{n,m \rightarrow \infty} m/n = f(k) > 2^k \ln(2)$ [CS88] (1988). The analysis shows the root cause to be a property known as sparseness; which roughly indicates the number of times pairs of clauses have a common literal or complementary pair of literals. When an expression is sparse any “moderately large” subset of its clauses must contain a “large” number of variables that occur exactly once in that subset. Sparsity forces the proof to have a large resolvent. But, almost all “short” resolution refutations contain no long clauses after eliminating all clauses satisfied by a particular small random partial assignment ρ . Moreover, resolution refutations for almost all unsatisfiable random (n, m, k) -CNF expressions with clauses satisfied by ρ removed are sparse and, therefore, must have at least one high width resolvent. Consequently, almost all unsatisfiable random expressions have long resolution refutations. Beame, Karp, Pitassi, Saks, Ben-Sasson, and Wigderson, among others, contributed ideas leading up to a concise understanding of this phenomenon in [BP96] (1996), [BKPS98] (1998), [BSW01] (2001).

Despite considerable tweaking, the best we can say right now is that, with probability tending to 0, the width of a shortest resolution proof is bounded

by a polynomial in n when $m/n > 2^k \ln(2)$ and $\lim_{m,n \rightarrow \infty} m/n^{(k+2)/(4-\epsilon)} < 1$, where ϵ is some small constant; and with probability tending to 1, the width is polynomially bounded when $\lim_{m,n \rightarrow \infty} m/n > (n/\log(n))^{k-2}$.

The failure of resolution has led to the development of a new technique by Goerdts for certifying unsatisfiability, mentioned above, that uses eigenvalues. In [GK01] this spectral technique is used to obtain bounds sufficient to show that certifying unsatisfiability in polynomial time can be accomplished with high probability when $\lim_{m,n \rightarrow \infty} m/n > n^{k/2-1+o(1)}$ which is considerably better than resolution.

1.22. Probabilistic analysis: thresholds

Results of the 1980s (see Section 1.21) showed that, for random width k expressions density, that is the ratio m/n , is correlated with certain interesting expression properties. For example, it was shown that if $m/n > 2^k \ln(2)$ then a random instance is unsatisfiable with high probability and if $m/n < O(1) \cdot 2^k/k$ a random expression is satisfiable with high probability. So, it was clear that a crossover from probably unsatisfiable to probably satisfiable occurs as density is changed. Early on it was speculated that “hardness” is directly related to the nature of crossover and this motivated the study of Satisfiability thresholds.

Actually, thresholds can apply to a multitude of properties, not just the property of a formula being satisfiable, so we give a general definition of threshold here. Let $X = \{x_1, \dots, x_e\}$ be a set of e elements. Let A_X , a subset of the power set of X (denoted 2^X), be called a *property*. Call A_X a *monotone property* if for any $s \in A_X$, if $s \subset s'$, then $s' \in A_X$. Typically, a monotone property follows from a high-level description which applies to an infinite family of sets X . For example, let $X = \mathcal{C}_{k,n}$ be the set of all non-tautological, width k clauses that can be constructed from n variables. Thus $e = 2^k \binom{n}{k}$ and any $s \in 2^{\mathcal{C}_{k,n}}$ is a k -CNF expression. Let $UNSAT_{\mathcal{C}_{k,n}}$ denote the property that a k -CNF expression constructed from n variables is unsatisfiable. That is, any $s \in UNSAT_{\mathcal{C}_{k,n}}$ has no model and any $s \in 2^{\mathcal{C}_{k,n}} \setminus UNSAT_{\mathcal{C}_{k,n}}$ has a model. If $s \in UNSAT_{\mathcal{C}_{k,n}}$ and $c \in \mathcal{C}_{k,n}$ such that $c \notin s$, then $s \cup \{c\} \in UNSAT_{\mathcal{C}_{k,n}}$ so the property $UNSAT_{\mathcal{C}_{k,n}}$ is monotone for $k < n$.

For any $0 \leq p \leq 1$ and any monotone property $A_X \subset 2^X$ define

$$\mu_p(A_X) = \sum_{s \in A_X} p^{|s|} (1-p)^{e-|s|}$$

to be the probability that a random set has the monotone property. For the property $UNSAT_{\mathcal{C}_{k,n}}$ (among others), s is a set of clauses, hence this probability measure does not match that for what we call random k -CNF expressions but comes very close with $p = m/(2^k \binom{n}{k}) \approx m \cdot k!/(2n)^k$.

Observe that $\mu_p(A_X)$ is an increasing function of p .⁵ Let $p_c(X)$ denote that value of p for which $\mu_p(A_X) = c$. The values of $p_c(X)$ change as the size of X increases. There are two threshold types.

⁵By illustration using $UNSAT_{\mathcal{C}_{k,n}}$ this reflects the fact that, as p increases, the average number of clauses increases, so the probability that an expression has the $UNSAT_{\mathcal{C}_{k,n}}$ property increases.

A_X is said to have a *sharp threshold* if, for any small, positive ϵ ,

$$\lim_{|X| \rightarrow \infty} (p_{1-\epsilon}(X) - p_\epsilon(X))/p_{1/2}(X) = 0.$$

A_X is said to have a *coarse threshold* if, for any small, positive ϵ ,

$$\lim_{|X| \rightarrow \infty} (p_{1-\epsilon}(X) - p_\epsilon(X))/p_{1/2}(X) > 0.$$

From experiments based on the constant width distribution it appeared that $UNSAT_{C_{k,n}}$ has a sharp threshold and that the density at which the crossover occurs is a function of k . The so-called *satisfiability threshold* was therefore denoted by r_k . In addition, as m/n is reduced significantly below (the experimentally determined) r_k , algorithms of various types were observed to perform better and expressions were more likely to be members of a polynomial time solvable class (see Section 1.19 for a description - an analysis of these classes is given later in this section), and as m/n is increased significantly above r_k various algorithms were observed to perform better. But, in the neighborhood of r_k random expressions seemed to reach a point of maximum difficulty, at least for well-known, well-tried algorithms. Hence, the study of satisfiability thresholds was motivated by a possible connection between the nature of hardness and the nature and location of thresholds. The constant width distribution has dominated this study, perhaps because expressions generated when parameters are set near r_k tend to be extremely difficult for known algorithms.

The conjecture of a sharp threshold can be traced to a paper by Mitchell, Selman, and Levesque [MSL92] (1992) where the “easy-hard-easy” phenomenon is pointed out. A possible explanation is given by Cheeseman in [CKT91] (1991) where long “backbones” or chains of inference are stated to be a likely cause as the high density of well-separated “near-solutions” induced by backbones leads to thrashing in search algorithms. Tremendous insight on this phenomenon came from the field of Statistical Mechanics [MZK⁺99b, MZ02] where the connection between backbones and hardness was seen to be analogous to phase transitions in spin glasses. Achlioptas advanced this further with a rigorous analysis as this volume was going to press.

The satisfiability threshold for random $(n, m, 2)$ -CNF expressions was found by Chvátal and Reed to be sharp with $r_2 = 1$ [CR92] (1992) (it is historically correct to note that Fernandez de la Vega [FdlV92] and Goerdt [Goe96] independently achieved the same result in the same year). The fact that $(n, m, 2)$ -CNF expressions can be solved in polynomial time [Coo71] means that there is a *simple* characterization for those instances which are unsatisfiable. Both [CR92] and [Goe96] make use of this characterization by focusing on the emergence of the “most likely” unsatisfiable random $(n, m, 2)$ -CNF expressions.

For $k > 2$ the situation was much more difficult. The results of Section 1.21, some of which were bounded probability results, can be regarded as a history of lower bounds for the satisfiability threshold. Upper bounds were improved at a consistent rate for a while, the most intense investigation focusing on the case $k = 3$. Results due to Frieze and Suen [FS96] (1996), Kamath, Motwani, Palem, Spirakis [KMPS95] (1995), Fernandez de la Vega [EMFdlV97] (1997), Dubois,

Boufkhad [DB97] (1997), Kirousis, Kranakis, Krizanc [KKK96] (1996), Kirousis, Kranakis, Krizanc, Stamatiou [KKKS98] (1998), Dubois [Dub01] (2001), and Dubois, Boufkhad, Mandler [DBM00] (2002), containing many beautiful ideas, have brought the upper bound down to just over 4.5 for $k = 3$. From Section 1.21 the lower bound for $k = 3$ is 3.52. From experiments, we expect $r_3 \approx 4.25$.

Friedgut [Fri99] (1999) proved that sharp satisfiability thresholds exist for random (n, m, k) -CNF expressions, thereby confirming the conjecture of [MSL92]. This result immediately lifted all previously known and future constant probability bounds, to almost surely bounds. Friedgut's result has left open the possibility that the satisfiability threshold is a function of both k and n and it is still not known whether the satisfiability threshold depends on n , as weak as that dependency must be.

Monasson [MZK⁺99b, MZK⁺99a] (1999) and others conjectured that there is a strong connection between the “order” of threshold sharpness, that is whether the transition is smooth or discontinuous, and hardness. Consistent with this, using the characterization of unsatisfiable $(n, m, 2)$ -CNF expressions mentioned above, Bollobas et al. [BBC⁺01] (2001) completely determined the “scaling window” for random $(n, m, 2)$ -CNF expressions, showing that the transition from satisfiability to unsatisfiability occurs for $m = n + \lambda n^{2/3}$ as λ goes from $-\infty$ to $+\infty$. For some time scaling windows for various problems were consistent with Monasson's conjecture (e.g. [CD99, CD03b]). But eventually the conjecture was disproved in [ACIM01] (2001) where it was found that the order of threshold sharpness for the 1-in- k SAT problem, which is \mathcal{NP} -complete, is the same as that of 2-SAT.

On the other hand, the work of Creignou and Daudé [CD03a] (2002), [CD04] (2004) revealed the importance of *minimal monotonic structures* to the existence of sharp transitions. An element $s \in A_X$ is said to be *minimal* if for all $s' \subset s$, $s' \in 2^X \setminus A_X$. Those results have been used, for example, to show the limitations of most succinctly defined polynomial-time solvable classes of expressions, such as those mentioned in Section 1.19. Using density m/n of (n, m, k) -CNF distributions as a measure, thresholds for some classes of Section 1.19 have been determined: for example, a random (n, m, k) -CNF expression, $k \geq 3$, is q-Horn with probability tending to 1 if $m/n < O(1)/(k^2 - k)$ and with probability tending to 0 if $m/n > O(1)/(k^2 - k)$ [FVG03, CDF05]. Except for the matched class, the monotonic structure analysis shows why the classes of Section 1.19, including q-Horn, SLUR, renameable Horn and many others, are weak: they are “vulnerable” to cyclic clause structures, the presence of any of these in an expression prevents it from having the polynomial-time solvable property. A random expression is matched with probability tending to 1 if $m/n < 0.64$ [FVG03]: a result that adds perspective to the scope of most polynomial-time solvable classes.

Also of historical importance are results on $2 + p$ mixed random expressions: random expressions with width 2 and width 3 clauses, the p being the fraction of width 3 clauses. This distribution was introduced in [KS94] (1994) to help understand where random expressions get hard during search: if a search algorithm that embodies unit propagation is presented with a $(n, m, 3)$ -CNF expression, every search node represents such a mixed expression with a particular value of p so hardness for some range of p could translate to hardness for search. Experi-

mental evidence suggested that for some p_c , figured to be around 0.417, if $p < p_c$ the width 3 clauses were effectively irrelevant to the satisfiability of a random expression but if $p > p_c$ the transition behavior was more like that of a random width 3 expression. In [AKKK01] (2001) it was shown that $0.4 < p_c < 0.696$ and conjectured that $p_c = 0.4$. In [ABM04] (2004) it was shown that a random $2 + p$ mixed expression has a minimum exponential size resolution refutation (that includes any DPLL algorithm) with probability $1 - o(1)$ when the number of width 2 clauses is less than ρn , $\rho < 1$, and p is any constant. Actually, the results of this paper are quite far-ranging and provide new insights into the behavior of DPLL algorithms as they visit search nodes that represent $2 + p$ expressions.

1.23. Stochastic Local Search

by Holger Hoos

Stochastic local search (SLS) is one of the most successfully and widely used general strategies for solving hard combinatorial problems. Early applications to optimization problems date back to the 1950s (for example, see [Cro58, Flo56, Lin65]), and the Lin-Kernighan algorithm for the Traveling Salesman problem [LK73] (1973) is still among the most widely known problem-specific SLS algorithms. Two of the most prominent general SLS methods are Simulated Annealing [KGV83] (1983) and Evolutionary Algorithms [FOW66, Hol75, Sch81] (1966–1981).

SLS algorithms for SAT were first presented in 1992 by Gu [Gu92] and Selman et al. [SLM92] (following earlier applications to Constraint Satisfaction [MJPL90] and MAX-SAT [HJ90]). Interestingly, both Gu and Selman et al. were apparently unaware of the MAX-SAT work, and Hansen and Jaumard and Minton et al. appear to have been unaware of each other's work. The success of Selman et al.'s GSAT algorithm in solving various types of SAT instances more effectively than DPLL variants of the day sparked considerable interest in the AI community, giving rise to a fruitful and active branch of SAT research. GSAT is based on a simple iterative best improvement method with static restarts; in each local search step, it flips the truth value of one propositional variable such that the number of unsatisfied clauses in the given CNF formula is maximally reduced.

Within a year, the original GSAT algorithm was succeeded by a number of variants. These include HSAT [GW93], which uses a very limited form of search memory to avoid unproductive cycling of the search process; GSAT with Clause Weighting [SK93], which achieves a similar goal using dynamically changing weights associated with the clauses of the given CNF formula; and GSAT with Random Walk (GWSAT) [SK93], which hybridizes the “greedy” search method underlying GSAT with a simple random walk procedure (which had previously been shown by Papadimitriou [Pap91] to solve satisfiable 2-CNF formulae almost certainly in $O(n^2)$ steps).

Two relatively subtle modifications of GWSAT lead to the prominent (basic) WalkSAT algorithm [SKC94], which is based on the idea of selecting a currently unsatisfied clause in each step and satisfying that clause by flipping the value assigned to one of its variables. Basic WalkSAT (also known as WalkSAT/SKC) was shown empirically to outperform GWSAT and most other GSAT variants

for a broad range of CNF formulae; it is also somewhat easier to implement. Variants of WalkSAT that additionally use search memory, in particular WalkSAT/Tabu [MSK97] (1997) and Novelty⁺ [Hoo98, Hoo99] (1998) – an extension of the earlier Novelty algorithm of McAllester et al. [MSK97] – typically achieve even better performance. Novelty+, which has been proven to solve satisfiable CNF formulae with arbitrarily high probability given sufficient time, was later extended with an adaptive noise mechanism [Hoo02] (2002), and the resulting Adaptive Novelty+ algorithm is still one of the most effective SLS algorithms for SAT currently known.

Based on the same fundamental idea of dynamically changing clause weights as GSAT with Clause Weighting, Wah et al. developed an approach known as Discrete Lagrangian Method [SW98] (1998), whose later variants were shown to be highly effective, particularly for solving structured SAT instances [WW99, WW00]. A conceptually related approach, which uses multiplicatively modified clause weights has been developed by Schuurmans et al. [SSH01] (2001) and later improved by Hutter et al. [HTH02] (2002), whose SAPS algorithm was, until recently, one of the state-of-the-art SLS algorithms for SAT.

A detailed survey of SLS algorithms for SAT up to 2004 can be found in Chapter 6 of the book by Hoos and Stützle [HS05]. Since then, a number of new algorithms have been developed, including Li and Huang's G²WSAT [LH05] (2005), which combines ideas from the GSAT and WalkSAT family of algorithms, and Ishtaiwi et al.'s DDFW algorithm [ITA⁺06] (2006), which uses a dynamic clause weight redistribution. In another line of work, Anbulagan et al. have reported results suggesting that by using a resolution-based preprocessing method, the performance of several state-of-the-art SLS algorithms for SAT can be further improved [APSS05](2005).

SLS algorithms for SAT have also played an important role in theoretical work on upper bounds on worst-case time complexity for solving SAT on k -CNF formulae; this includes the previously mentioned random walk algorithm by Papadimitriou [Pap91] (1991) and later extensions by Schöning [Sch99] (1999) and Schuler et al. [SSW01] (2001).

1.24. Maximum Satisfiability

by Hantao Zhang

The problem of finding a truth assignment to the variables of a CNF expression that satisfies the maximum number of clauses possible is known as Maximum Satisfiability or MAX-SAT. If clauses have at most two literals each, the problem is known as MAX-2-SAT. The decision version of MAX-SAT and even MAX-2-SAT is \mathcal{NP} -complete. Unfortunately, there is no polynomial time approximation scheme for MAX-SAT unless $\mathcal{P} = \mathcal{NP}$ [AL97]. Because the MAX-SAT problem is fundamental to many practical problems in Computer Science [HJ90] and Electrical Engineering [XRS02], efficient methods that can solve a large set of instances of MAX-SAT are eagerly sought.

1.24.1. MAX-SAT: Decision Algorithms

Many of the proposed methods for MAX-SAT are based on approximation algorithms [DGH⁺02] (2002); some of them are based on branch-and-bound methods [HJ90] (1990), [BF99] (1999), [BR99] (1999), [Hir00] (2000), [NR00] (2000), [dGLMS03] (2003); and some of them are based on transforming MAX-SAT into SAT [XRS02] (2002), [ARMS02a] (2002).

Worst-case upper bounds have been obtained with respect to three parameters: the length L of the input formula (*i.e.*, the number of literals in the input), the number m of the input's clauses, and the number n of distinct variables occurring in the input. The best known bounds for MAX-SAT are $O(L2^{m/2.36})$ and $O(L2^{L/6.89})$ [BR99] (1999). The question of whether there exist exact algorithms with complexity bounds down to $O(L2^n)$ has been open and of great interest (see [Nie98] (1998), [AGN01] (2000), [GHNR03] (2003)) since an algorithm which enumerates all the 2^n assignments and then counts the number of true clauses in each assignment would take time $O(L2^n)$. Recently, it has been shown that a branch-and-bound algorithm can achieve $O(b2^n)$ complexity where b is the maximum number of occurrences of any variable in the input. Typically, $b \simeq L/n$.

The operation of the best branch-and-bound algorithms for MAX-SAT is similar to that of DPLL. Notable implementations are due to Wallace and Freuder (implemented in Lisp) [WF96] (1996), Gramm [Gra99] (1999), Borchers and Furman [BF99] (1999 - implemented in C and publicly available), Zhang, Shen, and Manyà [ZSM03] (2003), and Zhao and Zhang [ZZ04] (2004).

1.24.2. MAX-2-SAT: Decision Algorithms

MAX-2-SAT is important because a number of other \mathcal{NP} -complete problems can be reduced to it in a natural way, for example graph problems such as Maximum Cut and Independent Set [CCTW96] (1996), [MR99a]. For MAX-2-SAT, the best bounds have been improved from $O(m2^{m/3.44})$ [BR99] (1999), to $O(m2^{m/2.88})$ [NR00] (2000), and recently to $O(m2^{m/5})$ [GHNR03] (2003). The recent branch-and-bound algorithm cited above results in a bound of $O(n2^n)$ since $b \leq 2n$. When $m = 4n^2$ the bound is $O(\sqrt{m}1.414^{\sqrt{m}})$, which is substantially better than the result reported in [GHNR03].

For random 2-CNF formulas satisfiability thresholds have been found as follows:

Theorem [CGHS04] :

1. For $c < 1$, $K(n, cn) = \Theta(1/n)$.
2. For c large,

$$(0.25c - 0.343859\sqrt{c} + O(1))n \geq K(n, cn) \geq (0.25c - 0.509833\sqrt{c})n.$$

3. For any fixed $\epsilon > 0$, $\frac{1}{3}\epsilon^3 n \geq K(n, (1 + \epsilon)n)$.

In the above theorem, \geq is a standard asymptotic notation: $f(n) \geq g(n)$ means that f is greater than or equal to g *asymptotically*, that is, $f(n)/g(n) \geq 1$ when n goes to infinity, although it may be that $f(n) < g(n)$ even for arbitrarily large values of n .

1.25. Nonlinear formulations

by Miguel Anjos

The nonlinear formulations for SAT are based on the application of the fundamental concept of lift-and-project for constructing tractable continuous relaxations of hard binary (or equivalently, Boolean) optimization problems. The application of continuous relaxations to binary optimization dates back at least to Lovász's introduction of the so-called theta function as a bound for the stability number of a graph [Lov79]. More generally, the idea of liftings for binary optimization problems has been proposed by several researchers, and has led to different general-purpose frameworks. Hierarchies based on linear programming relaxations include the lift-and-project method of Balas, Ceria and Cornuéjols [BCC93], the reformulation-linearization technique of Sherali and Adams [SA90], and the matrix-cuts approach of Lovász and Schrijver [LS91]. Researchers in the SAT community have studied the complexity of applying some of these techniques, and generalizations thereof, to specific classes of SAT problems (see the recent papers [BOGH⁺03, GHP02a, GHP02b]).

While the aforementioned techniques use linear programming relaxations, the recent Lasserre hierarchy is based on semidefinite programming relaxations [Las00, Las02]. (Semidefinite constraints may also be employed in the Lovász-Schrijver matrix-cuts approach, but in a different manner from that of the Lasserre paradigm.) A detailed analysis of the connections between the Sherali-Adams, Lovász-Schrijver, and Lasserre frameworks was done by Laurent [Lau03]. In particular, Laurent showed that the Lasserre framework is the tightest among the three.

Semidefinite programming (SDP) refers to the class of optimization problems where a linear function of a symmetric matrix variable X is optimized subject to linear constraints on the elements of X and the additional constraint that X must be positive semidefinite. This includes linear programming problems as a special case, namely when all the matrices involved are diagonal. The fact that SDP problems can be solved in polynomial-time to within a given accuracy follows from the complexity analysis of the ellipsoid algorithm (see [GLS93]). A variety of polynomial time interior-point algorithms for solving SDPs have been proposed in the literature, and several excellent solvers for SDP are now available. The SDP webpage [Hel] and the books [dK02, WSVe00] provide a thorough coverage of the theory and algorithms in this area, as well as a discussion of several application areas where semidefinite programming researchers have made significant contributions. In particular, SDP has been very successfully applied in the development of approximation algorithms for several classes of hard combinatorial optimization problems, including maximum-satisfiability (MAX-SAT) problems.

A σ -approximation algorithm for MAX-SAT is a polynomial-time algorithm that computes a truth assignment such that at least a proportion σ of the clauses in the MAX-SAT instance are satisfied. The number σ is the approximation ratio

or guarantee. For instance, the first approximation algorithm for MAX-SAT is a $\frac{1}{2}$ -approximation algorithm due to Johnson [Joh74]: given n values $\pi_i \in [0, 1]$, the algorithm sets the i^{th} Boolean variable to 1 independently and randomly with probability π_i ; the resulting total expected weight of the satisfied clauses is $\frac{1}{2}$. Unless $\mathcal{P} = \mathcal{NP}$, there is a limit to the approximation guarantees that can be obtained. Indeed, Håstad [Hås01] proved that unless $\mathcal{P} = \mathcal{NP}$, for any $\epsilon > 0$, there is no $(\frac{21}{22} + \epsilon)$ -approximation algorithm for MAX-2-SAT, and no $(\frac{7}{8} + \epsilon)$ -approximation algorithm for MAX-SAT.

The most significant breakthrough was by Goemans and Williamson [GW95] who proposed an SDP-based 0.87856-approximation algorithm for MAX-2-SAT. The key to their analysis is the ingenious use of a randomly generated hyperplane to extract a binary solution from the set of n -dimensional vectors defined by the solution of the SDP relaxation. The randomized hyperplane rounding procedure can be formally de-randomized using the techniques in [MR99b].

A further significant improvement was achieved by Feige and Goemans [FG95] who proposed a 0.931-approximation algorithm for MAX-2-SAT. There are two key innovations introduced by Feige and Goemans. The first one is that they tighten the SDP relaxation of Goemans and Williamson by adding the $\binom{n}{3}$ triangle inequalities. From the optimal solution of this strengthened SDP relaxation, they similarly obtain a set of vectors, but instead of applying the random hyperplane rounding technique to these vectors directly, they use them to generate a set of rotated vectors to which they then apply the hyperplane rounding.

Karloff and Zwick [KZ97] proposed a general construction of SDP relaxations for MAX- k -SAT. Halperin and Zwick [HZ01] consider strengthened SDP relaxations for MAX- k -SAT, and specifically for MAX-4-SAT, they studied several rounding schemes, and obtained approximation algorithms that almost attain the theoretical upper bound of $\frac{7}{8}$. Most recently, Asano and Williamson [AW02] have combined ideas from several of the aforementioned approaches and obtained a 0.7846-approximation algorithm for general MAX-SAT.

For the decision version of SAT, the first SDP-based approach is the Gap relaxation of de Klerk, van Maaren, and Warners [dKvMW00, dKvM03]. This SDP relaxation was inspired by the work of Goemans and Williamson as well as by the concept of elliptic approximations for SAT instances. These approximations were first proposed in [vM99] and were applied to obtain effective branching rules as well as to recognize certain polynomially solvable classes of SAT instances. The idea behind the elliptic approximations is to reformulate a SAT formula on n boolean variables as the problem of finding a ± 1 (hence binary) n -vector in an intersection of ellipsoids in \mathcal{R}^n . Although it is difficult to work directly with intersections of ellipsoids, it is possible to relax the formulation to an SDP problem. The resulting SDP relaxation is called the Gap relaxation. This relaxation characterizes unsatisfiability for 2-SAT problems [dKvMW00]. More interestingly, it also characterizes satisfiability for a class of covering problems, such as mutilated chessboard and pigeonhole instances. Rounding schemes and approximation guarantees for the Gap relaxation, as well as its behaviour on $(2 + p)$ -SAT problems, are studied in [dKvM03].

An elliptic approximation uses a quadratic representation of SAT formulas. More powerful relaxations can be obtained by considering higher-degree poly-

mial representations of SAT formulas. The starting point is to define for each clause a polynomial in ± 1 variables that equals 0 if and only if the clause is satisfied by the truth assignment represented by the values of the binary variables. Thus, testing satisfiability of a SAT formula is reduced to testing whether there are values $x_1, \dots, x_n \in \{-1, 1\}$ such that for every clause in the instance, the corresponding polynomial evaluated at these values equals zero.

We present two ways that SDP can be used to attempt to answer this question. One of them applies the Lasserre hierarchy mentioned above as follows. The Gap relaxation has its matrix variable in the space of $(n+1) \times (n+1)$ symmetric matrices, and is thus a first lifting. To generalize this operation, we allow the rows and columns of the SDP relaxations to be indexed by subsets of the discrete variables in the formulation. These larger matrices can be interpreted as higher liftings. Applying directly the Lasserre approach to SAT, we would use the SDP relaxations Q_{K-1} (as defined in [Las00]) for $K = 1, 2, \dots, n$ where the matrix variable of Q_{K-1} has rows and columns indexed by all the subsets I with $\|I\| \leq K$ (hence for $K = 1$, we obtain the matrix variable of the Gap relaxation). The results in [Las02] imply that for $K = n$, the resulting SDP relaxation characterizes satisfiability for every instance of SAT. However, this SDP has dimension exponential in n . Indeed, the SDP problems quickly become far too large for practical computation. This limitation motivated the study of partial higher liftings, where we consider SDP relaxations that have a much smaller matrix variable, as well as fewer linear constraints. The construction of such partial liftings for SAT becomes particularly interesting if we let the structure of the SAT instance specify the structure of the SDP relaxation.

One of these partial liftings was proposed in [Anj05]. This construction considers all the monomials $\prod_i x_i$ that appear in the instance's satisfiability conditions. An appropriate SDP relaxation is then defined where each row and column of the matrix variable corresponds to one of these terms. The resulting matrix is highly structured, and hence the SDP relaxation can be strengthened by adding some constraints that capture this structure. The tradeoff involved in adding such constraints to the SDP problem is that as the number of constraints increases, the SDP problems become increasingly more demanding computationally. Anjos [Anj05] defines the SDP relaxation R_3 by proposing to add a relatively small number of these constraints, judiciously chosen so that it is possible to prove the following result: if R_3 is infeasible, then the SAT instance is unsatisfiable; while if R_3 is feasible, and Y is a feasible matrix such that $\text{rank}(Y) \leq 3$, then the SAT instance is satisfiable, and a model can be extracted from Y . Thus the SDP relaxation can prove either satisfiability or unsatisfiability of the given SAT instance. A more compact relaxation is obtained by defining the columns of the matrix variable using only the sets of odd cardinality. This yields the SDP relaxation R_2 [Anj04a], an intermediate relaxation between the Gap relaxation (call it R_1) and R_3 . The names of the relaxations reflect their increasing strength in the following sense: For $k = 1, 2, 3$, any feasible solution to the relaxation R_k with rank at most k proves satisfiability of the corresponding SAT instance. Furthermore, the increasing values of k also reflect an improving ability to detect unsatisfiability, and an increasing computational time for solving the relaxation.

From the computational point of view, it is only possible to tackle relatively

small SAT instances (regardless of the choice of SDP relaxation) if branching is needed [Anj04b]. However, when it does not require branching, the SDP approach can be competitive. For instance, the SDP approach can successfully prove (without branching) the unsatisfiability of the **hgen8** instances, one of which was the smallest unsatisfiable instance that remained unsolved during the SAT competitions of 2003 and 2004.

A second way to test whether there is a set of ± 1 values for which the clause polynomials all equal zero was proposed by van Maaren and van Norden [vMvN05, vMvN08]. They consider (among others) the aggregate polynomial obtained by summing all the polynomials arising from clauses. This polynomial turns out to be non-negative on $\{-1, 1\}^n$, and for $x \in \{-1, 1\}^n$ it equals the number of unsatisfied clauses. (Hence, MAX-SAT is equivalent to the minimization of this polynomial over $\{-1, 1\}^n$.) An SDP relaxation is obtained as follows. Suppose we are given a column vector β of monomials in the variables x_1, \dots, x_n and a polynomial $p(x)$. Then $p(x)$ can be written as a sum-of-squares (SOS) in terms of the elements of β if and only if there exists a matrix $S \succeq 0$ such that $\beta^T S \beta = p$ [Par03]. If S is symmetric positive semidefinite, then $S = W^T W$ for some matrix W , and hence we have an explicit decomposition of p as an SOS: $\beta^T S \beta = p \Rightarrow \|W\beta\|_2^2 = p$. The resulting SDP problem is

$$\begin{aligned} \max \quad & g \\ \text{s.t.} \quad & F_{\Phi}^{\mathcal{B}}(x) - g \equiv \beta^T S \beta \text{ modulo } I_{\mathcal{B}} \\ & S \succeq 0 \end{aligned}$$

where $I_{\mathcal{B}}$ denotes the ideal generated by the polynomials $x_k^2 - 1, k = 1, \dots, n$. (The fact that each degree k polynomial that is non-negative on $\{-1, 1\}^n$ can be expressed as an SOS modulo $I_{\mathcal{B}}$ follows from the work of Putinar [Put93].) Note that since β is fixed, the equation $F(x) - g = \beta^T S \beta$ is linear in S and g , and hence this is an SDP problem. The SOS approach can thus be applied to obtain proofs of (un)satisfiability. For instance, it is straightforward to prove that if there exists a monomial basis β and an $\epsilon > 0$ such that $F^{\mathcal{B}}(x) - \epsilon$ is a SOS modulo $I_{\mathcal{B}}$, then the underlying SAT formula is unsatisfiable.

For the SOS approach, different choices of the basis β result in different SDP relaxations. Among the choices considered by van Maaren and van Norden are the following: SOS_{GW} is the relaxation obtained using the basis containing $1, x_1, \dots, x_n$; SOS_p is obtained using the basis containing $1, x_1, \dots, x_n$, plus the monomial $x_{k_1} x_{k_2}$ for each pair of variables that appear together in a clause; SOS_{ap} is obtained using the basis containing $1, x_1, \dots, x_n$, plus the monomials $x_{k_1} x_{k_2}$ for all pairs of variables; SOS_t is obtained using the basis containing $1, x_1, \dots, x_n$, plus the monomial $x_{k_1} x_{k_2} x_{k_3}$ for each triple of variables that appear together in a clause; and SOS_{pt} is obtained using the basis containing $1, x_1, \dots, x_n$, plus the monomial $x_{k_1} x_{k_2}$ for each pair of variables that appear together in a clause, plus $x_{k_1} x_{k_2} x_{k_3}$ for each triple of variables that appear together in a clause.

The notation SOS_{GW} is justified by the fact that SOS_{GW} is precisely the dual of the SDP relaxation used by Goemans and Williamson in their seminal paper [GW95]. van Maaren and van Norden prove that SOS_{GW} gives the same upper bound for MAX-2-SAT as the relaxation of Goemans and Williamson. They also show that for each triple $x_{k_1} x_{k_2} x_{k_3}$, adding the monomials $x_{k_1} x_{k_2}$,

$x_{k_1}x_{k_3}$, and $x_{k_2}x_{k_3}$ gives an SDP relaxation at least as tight as that obtained by adding the corresponding triangle inequality to the Goemans-Williamson relaxation. Furthermore, they prove that the SDP relaxation SOS_{ap} is at least as tight as the Feige-Goemans relaxation, and that for every instance of MAX-3-SAT, the SDP relaxation SOS_{pt} provides a bound at least as tight as the Karloff-Zwick relaxation.

From the computational point of view, van Maaren and van Norden provide computational results comparing several of these relaxations on instances of varying sizes and varying ratios of number of clauses to number of variables. They propose rounding schemes for MAX-2-SAT and MAX-3-SAT based on SOS_p and SOS_t respectively, and present preliminary results comparing their performance with the rounding schemes mentioned above. They also compare the performance of the R_3 relaxation with the SOS approach using either SOS_t or SOS_{pt} . Their preliminary results suggest that SOS_{pt} offers the best performance.

The most recent result about the nonlinear approach is that the SDP approach can explicitly characterize unsatisfiability for the well-known Tseitin instances on toroidal grid graphs. Consider a $p \times q$ toroidal grid graph and for each node (i, j) , set the parameter $t(i, j) = 0$ or 1. Introduce a Boolean variable for each edge, and for each node (i, j) , define 8 clauses on the four variables adjacent to it as follows: if $t(i, j) = 0$, add all clauses with an odd number of negations; and if $t(i, j) = 1$, add all clauses with an even number of negations. It is clear that the SAT instance is unsatisfiable if and only if $\sum_{(i,j)} t(i, j)$ is odd. It is shown in [Anj06] how to construct an SDP relaxation with matrix variable of dimension $14pq$ and with $23pq - 1$ linear equality constraints such that the SDP problem is infeasible if and only if the SAT instance is unsatisfiable. Therefore, for these instances, the SDP-based approach provides, in theory, an explicit certificate of (un)satisfiability, and therefore makes it possible to numerically compute such a certificate to within a given precision in polynomial time.

1.26. Pseudo-Boolean Forms

Let \mathcal{B}^n be the set of binary vectors of length n . Mappings $f : \mathcal{B}^n \mapsto \mathfrak{R}$ are called *pseudo-Boolean* functions. Let $V = \{v_1, v_2, \dots, v_n\}$ be a set of n binary variables. Since there is a one-to-one correspondence between subsets of V and \mathcal{B}^n these functions are called set functions. There is a natural connection between pseudo-Boolean functions and binary optimization which has been exploited since the 1960s, especially following the seminal treatment of the topic by Hammer and Rudeanu in 1968 [HR68]. Areas impacted by pseudo-Boolean functions are diverse and include VLSI design (for example, [BGM88]), maximum satisfiability (for example, [Kar72]), clustering in statistics (for example, [Rao71]), economics (for example, [HS71]), graph theory (for example, [PR75]), and scheduling, among many others.

There are at least two important representations of pseudo-Boolean functions. All pseudo-Boolean functions may be uniquely represented as *multi-linear polynomials* of the form

$$f(v_1, v_2, \dots, v_n) = c_f + \sum_{S \subseteq V} c_S \prod_{v \in S} v$$

where c_S is some real number that depends on S and c_f is a constant. The degree of such a polynomial is the largest S for which $c_S \neq 0$. A degree 1 polynomial is called *linear*, a degree 2 polynomial is called *quadratic* and so on.

Polynomials can also be written with non-negative constants using products of literals instead of products of variables as follows:

$$f(v_1, v_2, \dots, v_n) = c_f + \sum_{S \subseteq V \cup \bar{V}} c_S \prod_{l \in S} l$$

where $\bar{V} = \{\neg v_1, \neg v_2, \dots, \neg v_n\}$ is the set of negative literals associated with variables (and positive literals) V and $c_S \geq 0$ with $c_S = 0$ if there is a complementary pair $\{v, \neg v\} \in S$. This is called the *posiform* representation of a pseudo-Boolean function. Although a posiform uniquely determines a pseudo-Boolean function, a pseudo-Boolean function may be represented by more than one posiform. Moreover, it is easy to compute a posiform from a polynomial representation but it may be difficult to compute a *unique* polynomial representation corresponding to a given posiform. Because factors c_S are non-negative, minimizing a posiform is essentially the same as maximizing the number of terms that are 0. Thus, the posiform representation is closely related to instances of maximum satisfiability and the problem of determining satisfiability of a Boolean formula may be viewed as testing whether the minimum of a posiform is equal to its constant term.

The optimization of a pseudo-Boolean function can be reduced in polynomial time to the optimization of a quadratic pseudo-Boolean function [Ros72] (1972) by repeatedly substituting a new variable for a product of two variables and the addition of constraints that force the new variable to take the value of the product. However, a posiform corresponding to a quadratic pseudo-Boolean function may have degree higher than 2.

The *basic algorithm* for finding the optimum value of a pseudo-Boolean function was introduced in [HRR63b, HRR63a] (1963) and refined in [HR68] (1968). Although the algorithm has exponential complexity, special cases have been found, for example where the algorithm is fixed-parameter tractable [CHJ90] (1990). An upper (lower) bound of a pseudo-Boolean function may be obtained by a term-by-term majorization (minorization) procedure which was introduced in [HHS84] (1984). An algorithm for posiforms known as the DDT algorithm was developed in [BHS89] (1989).

In [CEI96] (1996) a variant of the Gröbner basis algorithm was applied to systems of posiforms restricted to modulo 2 arithmetic. The base operations of that algorithms were posiform addition, modulo 2, and variable multiplication. Application of such an operation is called a derivation. It was shown in [CEI96] that the algorithm uses a number of derivations that is guaranteed to be within a polynomial of the minimum number possible and that the minimum number of derivations cannot be much greater than, and may sometimes be far less than, the minimum number needed by resolution.

More recently, conventional resolution-based SAT solvers have been generalized to solve systems of linear polynomials. An early example is OPBDP [Bar95] (1995). Notable advanced examples include PBS [ARMS02b] (2002) which has advanced to PBS4 [AAR] (2005), Galena [CK03] (2003), and Pueblo [SS06] (2006). In [ES06] (2006) it is shown that translating linear polynomials to CNF clauses

and then solving with a conventional SAT solver is a viable alternative to specialized pseudo-Boolean solvers although some expertise is needed to obtain the best translations. The authors point out that translations might be the best approach when problems are modeled with many clauses and a few pseudo-Boolean constraints.

1.27. Quantified Boolean formulas

by Hans Kleine Büning

The concept of quantified Boolean formulas (QBF) is an extension of propositional logic that allows existential (\exists) and universal (\forall) quantifiers. The intended semantics of quantified Boolean formulas, without free variables, is that a universally quantified formula $\psi = \forall x\phi$ is true if and only if for every assignment of the truth values 1 and 0 to the variable x , ψ is true. For existentially quantified formulas $\psi = \exists x\phi$, ψ is true if and only if there is an assignment to x for which ϕ is true. In the case of free variables, ψ is called *satisfiable* if there is a truth assignment to the free variables such that ψ is true. The satisfiability problem for quantified Boolean formulas is often denoted as QSAT.

Most of the research has been done for formulas in prenex form, that is, formulas of the form $Q_1x_1 \dots Q_nx_n\phi$, where $Q_i \in \{\exists, \forall\}$, x_1, \dots, x_n are variables, and ϕ is a propositional formula called the *matrix*. By standard techniques, every quantified Boolean formula can be transformed into a logically equivalent formula in prenex form. In the same way, by the well-known procedure for propositional formulas, logically equivalent formulas with a CNF matrix or 3-CNF matrix can be obtained. These classes are denoted as QCNF and Q3-CNF.

Quantified Boolean formulas with free variables are logically equivalent to Boolean functions. But, clearly, there is no polynomial p such that every n -ary Boolean function can be represented as a quantified Boolean formula of length $p(n)$. However, for various applications, QBFs lead to shorter formulas in comparison to propositional formulas. See, for example, [JB06].

The first papers on QBFs were motivated by questions arising from computational complexity. Like SAT for NP, it has been shown in [MS73] that QSAT is one of the prominent \mathcal{PSPACE} -complete problems. In a more detailed analysis, a strong relationship was shown between the satisfiability problem of formulas with a fixed number of alternations of quantifiers and the polynomial-time hierarchy [MS72] where the polynomial-time hierarchy is defined as follows ($k \geq 0$):

$$\begin{aligned} \Delta_0^P &:= \Sigma_0^P := \Pi_0^P := P \\ \Sigma_{k+1}^P &:= NP^{\Sigma_k^P}, \quad \Pi_{k+1}^P := co\Sigma_{k+1}^P, \quad \Delta_{k+1}^P := P^{\Sigma_k^P} \end{aligned}$$

For quantified Boolean formulas in prenex form, the prefix type is defined as follows:

1. The prefix type of a propositional formula is $\Sigma_0 = \Pi_0$.
2. Let Φ be a formula with prefix type Σ_n (Π_n respectively), then the formula $\forall x_1 \dots \forall x_n \Phi$ ($\exists x_1 \dots \exists x_n \Phi$ respectively) is of prefix type Π_{n+1} (Σ_{n+1} respectively).

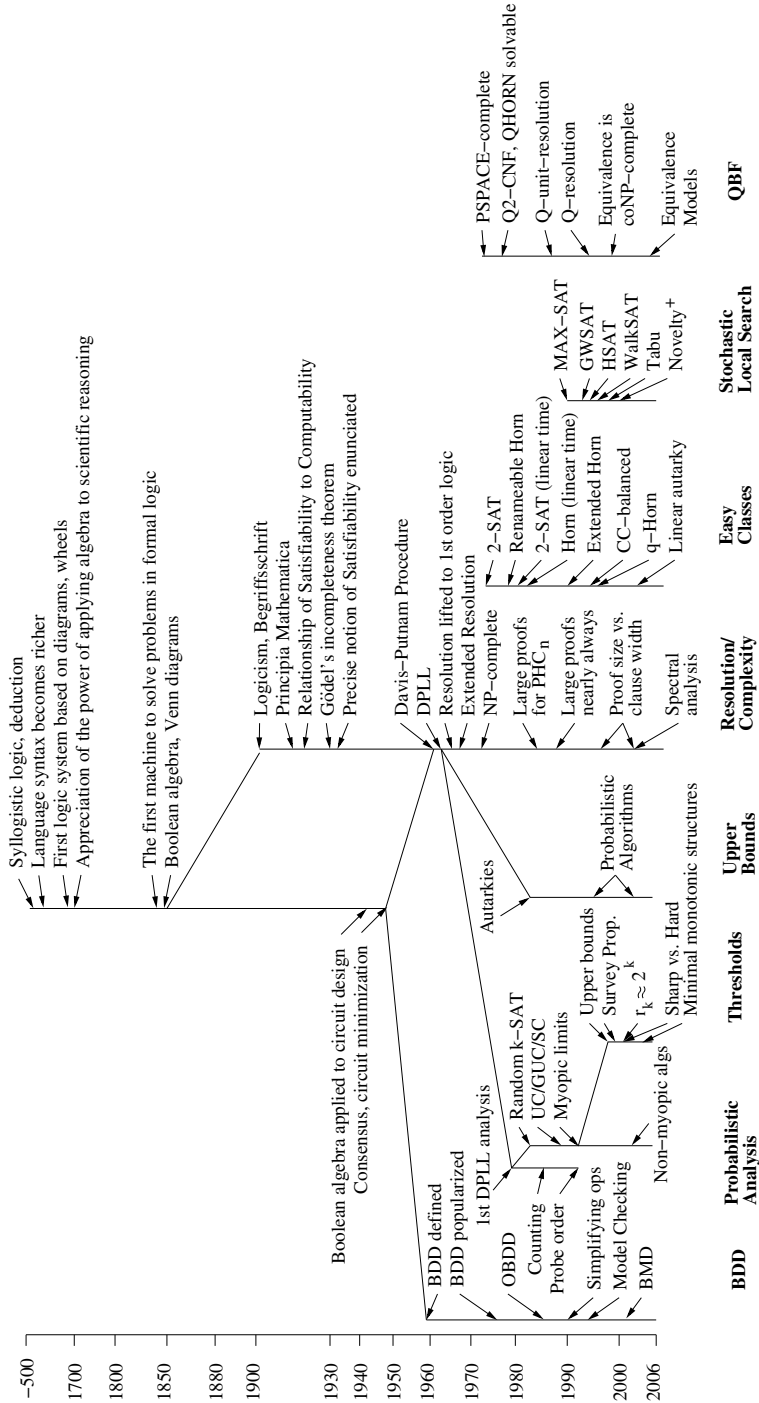


Figure 1.2. Timeline: Some nodes on the history tree (continued on next page). The intention is to provide some perspective on the level of activity in a particular area during a particular period.

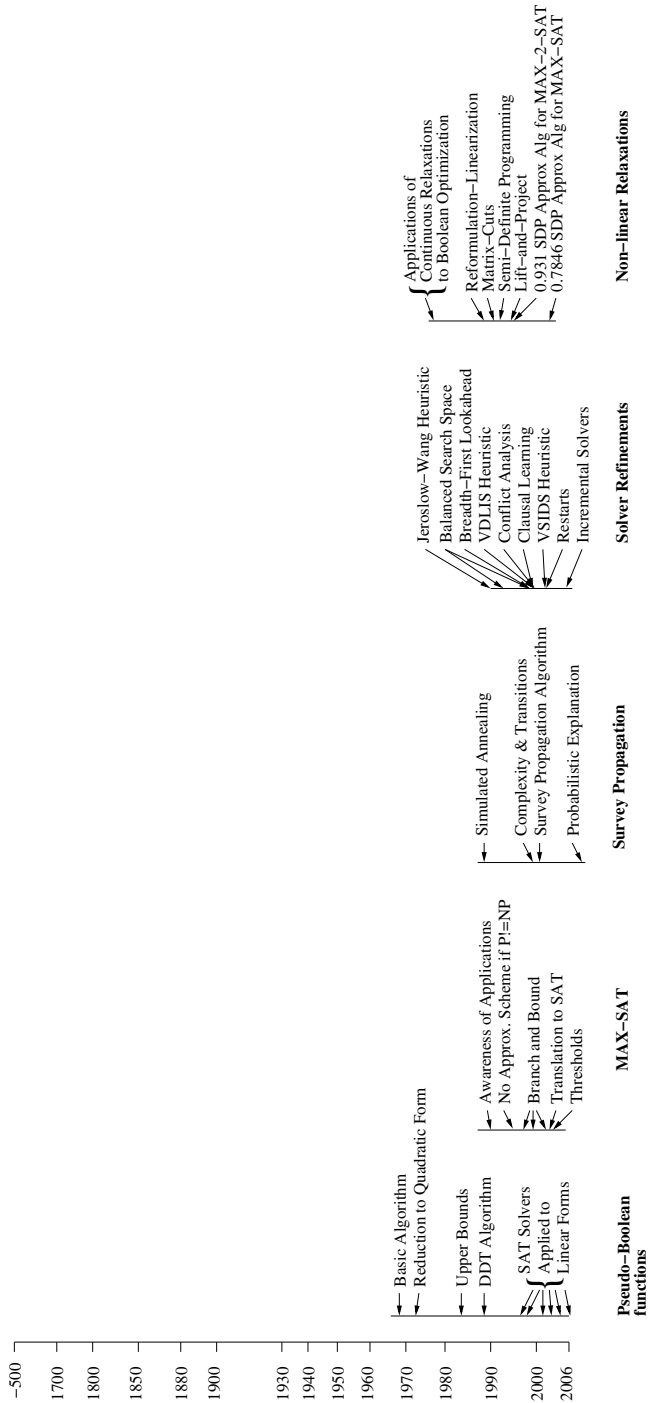


Figure 1.2. (continued) Survey propagation is shown on the chart but is not mentioned in the text. A history of algorithms inspired by statistical mechanics may be found in the Survey Propagation chapter of this Handbook.

It has been proved that for $k \geq 1$, the satisfiability problem for formulas with prefix type Σ_k (Π_k respectively) is Σ_k^P -complete (Π_k^P -complete respectively) [Sto77, Wra77]. Since $\mathcal{PSPACE} = \mathcal{NPSPACE}$ [Sav70], almost all quantified Boolean formula problems are solvable in \mathcal{PSPACE} . For example, in propositional logic, the equivalence problem is coNP -complete, whereas for quantified Boolean formulas, the problem remains \mathcal{PSPACE} -complete.

In addition to propositional formulas, a dichotomy theorem for quantified Boolean formulas has been established in [Sch73]. The idea was to classify classes of quantified Boolean formulas by means of a finite set of constraints, where the constraints are Boolean functions. The dichotomy theorem says that if the Boolean functions are equivalent to Horn formulas (anti-Horn formulas, 2-CNF formulas, XOR-CNF formulas respectively), then the satisfiability problems for the quantified classes are in P, otherwise they are \mathcal{PSPACE} -complete. As a consequence, the solvability of the satisfiability problem for Q2-CNF and QHORN follows, where QHORN is the set of formulas, whose matrix is a Horn formula. Detailed proofs and extensions of the dichotomy theorem can be found, for example, in [Dal97, CKS92]. For formulas with fixed prefix type, further results have been shown in [Hem94].

For Q2-CNF, the first linear-time algorithm for solving the satisfiability problem has been presented in [APT79]. For QHORN, the best known algorithm can be found in [KBKF95]. The latter algorithm requires not more than $O(r \cdot n)$ steps, where r is the number of universal variables and n is the length of the formula.

Not all the problems solvable in polynomial time for propositional formulas remain polynomial-time solvable for quantified formulas. For example, in contrast to the polynomial-time solvability of the equivalence problem for Horn formulas, the equivalence problem for quantified Horn formulas is coNP -complete [KBL99].

Instead of restrictions on the form of clauses, classes of quantified formulas satisfying some graph properties have been investigated. Examples are quantified versions of ordered binary decision diagrams (OBDDs) and free binary decision diagrams (FBDDs) (see, for example, [CMLBLM06]). For instance, the satisfiability problem for quantified FBDDs is \mathcal{PSPACE} -complete.

Q-resolution is an extension of the resolution calculus for propositional formulas to quantified formulas. Q-unit-resolution was introduced in [KKBS88] and was then generalized to Q-resolution in [KBKF95]. Here, a Q-unit clause contains at most one free or existentially quantified literal and arbitrarily many universal literals. The idea of Q-resolution is to resolve only over complementary pairs of existential or free variables, combined with a careful handling of the universal literals. Q-resolution is refutation complete and sound for QCNF. Similar to propositional Horn formulas, Q-unit-resolution is refutation complete for QHORN, and also for the class QEHORN. That is the set of formulas for which, after deleting the universal literals in the matrix, the remaining matrix is a Horn formula. The satisfiability problem for that class remains \mathcal{PSPACE} -complete [KKBF91].

A more functional view of the valuation of quantified Boolean formulas is the observation that a formula is true if and only if for every existential variable y there is a Boolean function $f_y(x_1, \dots, x_m)$ depending on the dominating universal variables x_1, \dots, x_m , such that after replacing the existential variables y by the associated functions $f_y(x_1, \dots, x_m)$ the formula is true. The set of such functions

is called a satisfiability model. For some classes of quantified Boolean formulas, the structure of the satisfiability models has been investigated. For example, satisfiable quantified Horn formulas have satisfiability models which consist of the constants *true* and *false* or conjunctions of variables. For Q2-CNF, the models are constants or a literal [KBSZ07]. Instead of satisfiability models, one can ask for Boolean functions, such that after replacing the existentially quantified variables with the functions, the formula is logically equivalent to the initial formula. These functions are called equivalence model. Equivalence models describe in a certain sense the internal dependencies of the formula. For example, quantified Horn formulas have equivalence models consisting of monotone functions. By means of this result, it has been shown that every quantified Horn formula can be transformed into an equivalent existentially quantified Horn formula in time $O(r \cdot n)$, where r is the number of universal variables and n is the length of the formula [BKBZ05].

References

- [AAR] F. Aloul and B. Al-Rawi. Pseudo-boolean solver, version 4. Available: <http://www.aloul.net/Tools/pbs4/>.
- [ABM04] D. Achlioptas, P. Beame, and M. Molloy. A sharp threshold in proof complexity yields lower bounds for satisfiability search. *Journal of Computer and System Sciences*, 68:238–268, 2004.
- [Ach00] D. Achlioptas. Setting 2 variables at a time yields a new lower bound for random 3-sat. In *32nd ACM Symposium on Theory of Computing*, pages 28–37. Association for Computing Machinery, New York, 2000.
- [Ach01] D. Achlioptas. Lower bounds for random 3-sat via differential equations. *Theoretical Computer Science*, 265:159–185, 2001.
- [ACIM01] D. Achlioptas, A. Chtcherba, G. Istrate, and C. Moore. The phase transition in 1-in- k sat and nae 3-sat. In *12th ACM-SIAM Symposium on Discrete Algorithms*, pages 721–722. Society for Industrial and Applied Mathematics, Philadelphia, 2001.
- [AGN01] J. Alber, J. Gramm, and R. Niedermeier. Faster exact algorithms for hard problems: a parameterized point of view. *Discrete Mathematics*, 229(1-3):3–27, 2001.
- [Ake78] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, 1978.
- [AKKK01] D. Achlioptas, L. M. Kirousis, E. Kranakis, and D. Krizanc. Rigorous results for random $(2 + p)$ -sat. *Theoretical Computer Science*, 265(1-2):109–129, 2001.
- [AL86] R. Aharoni and N. Linial. Minimal non-two-colorable hypergraphs and minimal unsatisfiable formulas. *Journal of Combinatorial Theory, Series A*, 43:196–204, 1986.
- [AL97] S. Arora and C. Lund. Hardness of approximation. In D. Hochbaum, editor, *Approximation algorithms for NP-hard problems*, chapter 10, pages 399–446. PWS Publishing Company, Boston, 1997.

- [AM02] D. Achlioptas and C. Moore. The asymptotic order of the random k -sat thresholds. In *43rd Annual Symposium on Foundations of Computer Science*, pages 779–788. IEEE Computer Society Press, Los Alamitos, CA, 2002.
- [Anj04a] M. F. Anjos. On semidefinite programming relaxations for the satisfiability problem. *Mathematical Methods of Operations Research*, 60(3):349–367, 2004.
- [Anj04b] M. F. Anjos. Proofs of unsatisfiability via semidefinite programming. In D. Ahr, R. Fahrion, M. Oswald, and G. Reinelt, editors, *Operations Research 2003*, pages 308–315. Springer-Verlag, Berlin, 2004.
- [Anj05] M. F. Anjos. An improved semidefinite programming relaxation for the satisfiability problem. *Mathematical Programming*, 102(3):589–608, 2005.
- [Anj06] M. F. Anjos. An explicit semidefinite characterization of satisfiability for tseitin instances on toroidal grid graphs. 2006.
- [AP04] D. Achlioptas and Y. Peres. The threshold for random k -sat is $2^k(\ln 2 + o(1))$. *Journal of the American Mathematical Society*, 17:947–973, 2004.
- [APSS05] Anbulagen, D. N. Pham, J. Slaney, and A. Sattar. Old resolution meets modern sls. In *12th National Conference on Artificial Intelligence (AAAI '05)*, pages 354–359. MIT Press, 2005.
- [APT79] B. Aspvall, M. F. Plass, and R. E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–132, 1979.
- [ARMS02a] F. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. Generic ilp versus specialized 0-1 ilp: An update. In *2002 International Conference on Computer-Aided Design (ICCAD '02)*, pages 450–457. IEEE Computer Society Press, Los Alamitos, CA, 2002.
- [ARMS02b] F. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. Pbs: a backtrack search pseudo-boolean solver. In *5th International Symposium on the Theory and Applications of Satisfiability Testing*. 2002. Available: http://www.aloul.net/Papers/faloul_sat02_pbs.pdf.
- [AS00] D. Achlioptas and G. Sorkin. Optimal myopic algorithms for random 3-sat. In *41st Annual Symposium on Foundations of Computer Science*, pages 590–600. IEEE Computer Society Press, Los Alamitos, CA, 2000.
- [Asp80] B Aspvall. Recognizing disguised nr(1) instances of the satisfiability problem. *Journal of Algorithms*, 1:97–103, 1980.
- [AW02] T. Asano and D. P. Williamson. Improved approximation algorithms for max sat. *Journal of Algorithms*, 42(1):173–202, 2002.
- [Bar69] M. E. Baron. A note on the historical development of logic diagrams. *The Mathematical Gazette: The Journal of the Mathematical Association*, 53(384):113–125, 1969.
- [Bar95] P. Barth. A davis-putnam based enumeration algorithm for linear pseudo-boolean optimization. Technical Report MPI-I-95-2-003,

- Max Plank Institut für Informatik, Saarbrücken, Germany, 1995.
- [BBC⁺01] B. Bollobás, C. Borgs, J. Chayes, J. H. Kim, and D. B. Wilson. The scaling window of the 2-sat transition. *Random Structures and Algorithms*, 18:201–256, 2001.
- [BC95] R. E. Bryant and Y.-A. Chen. Verification of arithmetic circuits with binary moment diagrams. In *32nd ACM/IEEE Design Automation Conference*, pages 535–541. IEEE Computer Society Press, Los Alamitos, CA, 1995.
- [BCC93] E. Balas, S. Ceria, and G. Cornuéjols. A lift-and-project cutting plane algorithm for mixed 0-1 programs. *Mathematical Programming*, 58(3, Ser. A):295–324, 1993.
- [BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, New York, 1999.
- [BCH90] E. Boros, Y. Crama, and P. L. Hammer. Polynomial-time inference of all valid implications for horn and related formulae. *Annals of Mathematics and Artificial Intelligence*, 1:21–32, 1990.
- [BCHS94] E. Boros, Y. Crama, P. L. Hammer, and M. Saks. A complexity index for satisfiability problems. *SIAM Journal on Computing*, 23:45–49, 1994.
- [BCM⁺92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.
- [BF99] B. Borchers and J. Furman. A two-phase exact algorithm for max-sat and weighted max-sat problems. *Journal of Combinatorial Optimization*, 2(4):299–306, 1999.
- [BFG⁺93] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *International Conference on Computer-Aided Design*, pages 188–191. IEEE Computer Society Press, Los Alamitos, CA, 1993.
- [BFU93] A. Z. Broder, A. M. Frieze, and E. Upfal. On the satisfiability and maximum satisfiability of random 3-cnf formulas. In *4th ACM-SIAM Symposium on Discrete Algorithms*, pages 322–330. Society for Industrial and Applied Mathematics, Philadelphia, 1993.
- [BGM88] F. Barahona, M. Grötschel, and A. R. Mahjoub. An application of combinatorial optimization to statistical physics and circuit layout design. *Operations Research*, 36:493–513, 1988.
- [BHS89] E. Boros, P. L. Hammer, and X. Sun. The ddt method for quadratic 0-1 optimization. Technical Report RRR 39-1989, Rutgers, the State University of New Jersey, 1989.
- [BHS94] E. Boros, P. L. Hammer, and X. Sun. Recognition of q-horn formulae in linear time. *Discrete Applied Mathematics*, 55:1–13, 1994.
- [BK04] T. Brueggemann and W. Kern. An improved local search algorithm for 3-sat. *Theoretical Computer Science*, 329:1–3, 2004.

- [BKB92] M. Buro and H. Kleine Büning. Report on a sat competition. Technical report, 1992.
- [BKBZ05] U. Bubeck, H. Kleine Büning, and X. Zhao. Quantifier rewriting and equivalence models for quantified horn formulas. In *8th International Conference on the Theory and Applications of Satisfiability Testing*, volume 3569 of *Lecture Notes in Computer Science*, pages 386–392. Springer, New York, 2005.
- [BKPS98] P. Beame, R. M. Karp, T. Pitassi, and M. Saks. On the complexity of unsatisfiability proofs for random k -cnf formulas. In *30th Annual Symposium on the Theory of Computing*, pages 561–571. Association for Computing Machinery, New York, 1998.
- [BKS03] P. Beame, H. A. Kautz, and A. Sabharwal. On the power of clause learning. In *18th International Joint Conference in Artificial Intelligence*, pages 94–99. Morgan Kaufmann Publishers, San Francisco, CA, 2003.
- [Bla37] A. Blake. *Canonical Expressions in Boolean Algebra*. PhD thesis, Department of Mathematics, University of Chicago, 1937.
- [BOGH⁺03] J. Buresh-Oppenheimer, N. Galesi, S. Hoory, A. Magen, and T. Pitassi. Rank bounds and integrality gaps for cutting plane procedures. In *44th Annual Symposium on Foundations of Computer Science*, pages 318–327. IEEE Computer Society Press, Los Alamitos, CA, 2003.
- [Bou76] R. T. Boute. The binary decision machine as a programmable controller. *EUROMICRO Newsletter*, 1(2):16–22, 1976.
- [BP96] P. Beame and T. Pitassi. Simplified and improved resolution lower bounds. In *37th Annual Symposium on Foundations of Computer Science*, pages 274–282. IEEE Computer Society Press, Los Alamitos, CA, 1996.
- [BR99] N. Bansal and V. Raman. Upper bounds for maxsat: Further improved. In *7th International Symposium on Algorithms and Computation*, volume 1741 of *Lecture Notes in Computer Science*, pages 247–258. Springer, New York, 1999.
- [BRB90] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a bdd package. In *27th ACM/IEEE Design Automation Conference*, pages 40–45. IEEE Computer Society Press, Los Alamitos, CA, 1990.
- [Bro03] F. M. Brown. *Boolean Reasoning*. Dover Publications, Mineola, New York, 2003.
- [Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
- [Bry92] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
- [BSIW04] E. Ben-Sasson, R. Impagliazzo, and A. Wigderson. Near optimal separation of tree-like and general resolution. *Combinatorica*, 24(4):585–603, 2004.
- [BSW01] E. Ben-Sasson and A. Wigderson. Short proofs are narrow - res-

- olution made simple. *Journal of the Association for Computing Machinery*, 48:149–169, 2001.
- [CBM90] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373. Springer, New York, 1990.
- [CCK⁺94] M. Conforti, G. Cornuéjols, A. Kapoor, K. Vučković, and M. R. Rao. Balanced matrices. In J.R. Birge and K.G. Murty, editors, *Mathematical Programming: State of the Art*. Braun-Brumfield, United States, 1994.
- [CCTW96] J. Cheriyan, W. H. Cunningham, L. Tuncel, and Y. Wang. A linear programming and rounding approach to max 2-sat. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 26, pages 395–414. DIMACS, Rutgers, the state University of New Jersey, 1996.
- [CD99] N. Creignou and H. Daudé. Satisfiability threshold for random xor-cnf formulas. *Discrete Applied Mathematics*, 96-97:41–53, 1999.
- [CD03a] N. Creignou and H. Daudé. Generalized satisfiability problems: minimal elements and phase transitions. *Theoretical Computer Science*, 302(1-3):417–430, 2003.
- [CD03b] N. Creignou and H. Daudé. Smooth and sharp thresholds for random k -xor-cnf satisfiability. *Theoretical Informatics and Applications*, 37(2):127–148, 2003.
- [CD04] N. Creignou and H. Daudé. Combinatorial sharpness criterion and phase transition classification for random csp. *Information and Computation*, 190(2):220–238, 2004.
- [CDF05] N. Creignou, H. Daudé, and J. Franco. A sharp threshold for the renameable horn and q-horn properties. *Discrete Applied Mathematics*, 153:48–57, 2005.
- [CEI96] M. Clegg, J. Edmonds, and R. Impagliazzo. Using the groebner basis algorithm to find proofs of unsatisfiability. In *28th Annual ACM Symposium on the Theory of Computing*, pages 174–183. Association for Computing Machinery, New York, 1996.
- [CF86] M.-T. Chao and J. Franco. Probabilistic analysis of two heuristics for the 3-satisfiability problem. *SIAM Journal on Computing*, 15:1106–1118, 1986.
- [CF90] M.-T. Chao and J. Franco. Probabilistic analysis of a generalization of the unit-clause literal selection heuristics for the k -satisfiability problem. *Information Sciences*, 51:289–314, 1990.
- [CGGT97] A. Cimatti, E. Giunchiglia, P. Giunchiglia, and P. Traverso. Planning via model checking: a decision procedure for ar. In *4th European Conference on Recent Advances in AI Planning (ECP '97)*, volume 1348 of *Lecture Notes in Computer Science*, pages 130–142. Springer, New York, 1997.
- [CGHS04] D. Coppersmith, D. Gamarnik, M. T. Hajiaghayi, and G. B. Sorkin. Random max sat, random max cut, and their phase tran-

- sitions. *Random Structures and Algorithms*, 24(4):502–545, 2004.
- [CH91] V. Chandru and J. N. Hooker. Extended horn sets in propositional logic. *Journal of the Association for Computing Machinery*, 38:205–221, 1991.
- [Cha84] R. Chandrasekaran. Integer programming problems for which a simple rounding type of algorithm works. In W. Pulleyblank, editor, *Progress in Combinatorial Optimization*, pages 101–106. Academic Press Canada, Toronto, Ontario, Canada, 1984.
- [CHJ90] Y. Crama, P. Hansen, and B. Jaumard. The basic algorithm for pseudo-boolean programming revisited. *Theoretical Computer Science*, 29(2-3):171–185, 1990.
- [Chu36] A. S. Church. Formal definitions in the theory of ordinal numbers. *Fundamental Mathematics*, 28:11–21, 1936.
- [CK03] D. Chai and A. Kuehlmann. A fast pseudo-boolean constraint solver. In *40th ACM/IEEE Design Automation Conference*, pages 830–835. IEEE Computer Society Press, Los Alamitos, CA, 2003.
- [CKS92] N. Creignou, S. Khanna, and M. Sudan. *Complexity classifications of Boolean constraint satisfaction problems*. Monographs on Discrete Applied Mathematics. Society for Industrial and Applied Mathematics, Philadelphia, 1992.
- [CKT91] P. Cheeseman, B. Kanefsky, and W. M. Taylor. Where the really hard problems are. In *12th International Joint Conference on Artificial Intelligence*, pages 331–340. Morgan Kaufmann Publishers, San Francisco, CA, 1991.
- [CM90] O. Coudert and J. C. Madre. A unified framework for the formal verification of sequential circuits. In *International Conference on Computer-Aided Design (ICCAD '90)*, pages 126–129. IEEE Computer Society Press, Los Alamitos, CA, 1990.
- [CMFT93] O. Coudert, J. C. Madre, H. Fraisse, and H. Touati. Implicit prime cover computation: an overview. In *1st Workshop on Synthesis And System Integration of Mixed Information technologies*. Nara, Japan, 1993.
- [CMLBLM06] S. Coste-Marquis, D. Le Berre, F. Letombe, and P. Marquis. Complexity results for quantified boolean formulae based on complete propositional languages. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:61–88, 2006.
- [Coo71] S. A. Cook. The complexity of theorem-proving procedures. In *3rd Annual ACM Symposium on Theory of Computing*, pages 151–158. Association for Computing Machinery, New York, 1971.
- [Cou94] O. Coudert. Two-level logic minimization: an overview. *Integration*, 17(2):97–140, 1994.
- [CR92] V. Chvátal and B. Reed. Mick gets some (the odds are on his side). In *33rd Annual Symposium on Foundations of Computer Science*, pages 620–627. IEEE Computer Society Press, Los Alamitos, CA, 1992.
- [Cro58] G. A. Croes. A method for solving traveling salesman problems. *Operations Research*, 6:791–812, 1958.

- [CS88] V. Chvátal and E. Szemerédi. Many hard examples for resolution. *Journal of the Association for Computing Machinery*, 35:759–768, 1988.
- [Dal97] V. Dalmau. Some dichotomy theorems on constant free boolean formulas. Technical Report TR-LSI-97-43-R, Universitat Polytechnica de Catalunya, 1997.
- [Dav01] M. Davis. The early history of automated deduction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. MIT Press, 2001.
- [DB97] O. Dubois and Y. Boufkhad. A general upper bound for the satisfiability threshold of random r -sat formulae. *Journal of Algorithms*, 24:395–420, 1997.
- [DBM00] O. Dubois, Y. Boufkhad, and J. Mandler. Typical random 3-sat formulae and the satisfiability threshold. In *11th ACM-SIAM Symposium on Discrete Algorithms*, pages 124–126. Society for Industrial and Applied Mathematics, Philadelphia, 2000.
- [DG84] W. F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming*, 1:267–284, 1984.
- [DGH⁺02] E. Dantsin, A. Goerdt, E. A. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan, and U. Schöning. A deterministic $(2 - 2/(k + 1))^n$ algorithm for k -sat based on local search. *Theoretical Computer Science*, 289:69–83, 2002.
- [dGLMS03] S. de Givry, J. Larrosa, P. Meseguer, and T. Schiex. Solving max-sat as weighted csp. In *9th International Conference on Principles and Practice of Constraint Programming (CP 2003)*, volume 2833 of *Lecture Notes in Computer Science*, pages 363–376. Springer, New York, 2003.
- [dK02] E. de Klerk. *Aspects of Semidefinite Programming*, volume 65 of *Applied Optimization*. Kluwer Academic Publishers, Dordrecht, 2002.
- [DK03] R. Damiano and J. Kukula. Checking satisfiability of a conjunction of bdds. In *40th ACM/IEEE Design Automation Conference*, pages 818–823. IEEE Computer Society Press, Los Alamitos, CA, 2003.
- [dKvM03] E. de Klerk and H. van Maaren. On semidefinite programming relaxations of $(2 + p)$ -sat. *Annals of Mathematics and Artificial Intelligence*, 37(3):285–305, 2003.
- [dKvMW00] E. de Klerk, H. van Maaren, and J. P. Warners. Relaxations of the satisfiability problem using semidefinite programming. *Journal of Automated Reasoning*, 24(1-2):37–65, 2000.
- [DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.
- [DP58] M. Davis and H. Putnam. Computational methods in the propositional calculus. unpublished report, Rensselaer Polytechnic Institute, 1958.
- [DP60] M. Davis and H. Putnam. A computing procedure for quantifica-

- tion theory. *Journal of the Association for Computing Machinery*, 7(3):201–215, 1960.
- [Dub01] O. Dubois. Upper bounds on the satisfiability threshold. *Theoretical Computer Science*, 265:187–197, 2001.
- [DW04] E. Dantsin and A. Wolpert. Derandomization of schuler’s algorithm for sat. In *6th International Conference on the Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 69–75. Springer, New York, 2004.
- [DW05] E. Dantsin and A. Wolpert. An improved upper bound for sat. In *8th International Conference on the Theory and Applications of Satisfiability Testing*, volume 3569 of *Lecture Notes in Computer Science*, pages 400–407. Springer, New York, 2005.
- [EIS76] S. Even, A. Itai, and A. Shamir. On the complexity of timetable and multi-commodity flow problems. *SIAM Journal on Computing*, 5:691–703, 1976.
- [EMFd1V97] A. El Maftouhi and W. Fernandez de la Vega. On random 3-sat. Laboratoire de Recherche en Informatique, Université Paris-Sud, France, 1997.
- [ES06] Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into sat. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–25, 2006.
- [Fd1V92] W. Fernandez de la Vega. On random 2-sat. Manuscript, 1992.
- [FG95] U. Feige and M. Goemans. Approximating the value of two prover proof systems, with applications to max 2sat and max dicut. In *3rd Israel Symposium on the Theory of Computing and Systems*, pages 182–189. IEEE Computer Society Press, Los Alamitos, CA, 1995.
- [FH89] J. Franco and Y. C. Ho. Probabilistic performance of heuristic for the satisfiability problem. *Discrete Applied Mathematics*, 22:35–51, 1988/89.
- [FKS⁺04] J. Franco, M. Kouril, J. Schlipf, J. Ward, S. Weaver, M. Dransfield, and W. M. Vanfleet. Sbsat: a state-based, bdd-based satisfiability solver. In *6th International Conference on the Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 398–410. Springer, New York, 2004.
- [Flo56] M. M. Flood. The traveling salesman problem. *Operational Research*, 4(1):61–75, 1956.
- [FOW66] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence Through Simulated Evolution*. John Wiley & Sons, New York, 1966.
- [FP83] J. Franco and M. Paull. Probabilistic analysis of the davis-putnam procedure for solving the satisfiability problem. *Discrete Applied Mathematics*, 5:77–87, 1983.
- [Fra86a] J. Franco. Elimination of infrequent variables improves average case performance of satisfiability algorithms. *SIAM Journal on Computing*, 20:103–106, 1986.
- [Fra86b] J. Franco. On the probabilistic performance of algorithms for the

- satisfiability problem. *Information Processing Letters*, 23:103–106, 1986.
- [Fra93] J. Franco. On the occurrence of null clauses in random instances of satisfiability. *Discrete Applied Mathematics*, 41:203–209, 1993.
- [Fre95] J. W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, USA, 1995.
- [Fri99] E. Friedgut. Sharp thresholds of graph properties, and the k -sat problem. *Journal of the American Mathematical Society*, 12(4):1017–1054, 1999. with an appendix by J. Bourgain.
- [FS96] A. M. Frieze and S. Suen. Analysis of two simple heuristics on a random instance of k -sat. *Journal of Algorithms*, 20:312–355, 1996.
- [FVG03] J. Franco and A. Van Gelder. A perspective on certain polynomial time solvable classes of satisfiability. *Discrete Applied Mathematics*, 125(2–3):177–214, 2003.
- [Gal77] Z. Galil. On the complexity of regular resolution and the Davis-Putnam procedure. *Theoretical Computer Science*, 4:23–46, 1977.
- [Gar58] M. Gardner. *Logic Machines and Diagrams*. McGraw-Hill, New York, 1958.
- [Gel59] H. Gelernter. Realization of a geometry-theorem proving machine. In *International Conference on Information Processing*, pages 273–282. UNESCO House, 1959.
- [GHN03] J. Gramm, E. A. Hirsch, R. Niedermeier, and P. Rossmanith. New worst-case upper bounds for max-2-sat with application to max-cut. *Discrete Applied Mathematics*, 130(2):139–155, 2003.
- [GHP02a] D. Grigoriev, E. A. Hirsch, and D. V. Pasechnik. Complexity of semialgebraic proofs. *Moscow Mathematics Journal*, 2(4):647–679, 2002.
- [GHP02b] D. Grigoriev, E. A. Hirsch, and D. V. Pasechnik. Exponential lower bound for static semi-algebraic proofs. In *29th International Colloquium on Automata, Languages and Programming (ICALP '02)*, volume 2380 of *Lecture Notes in Computer Science*, pages 257–268. Springer, New York, 2002.
- [Gil60] P. Gilmore. A proof method for quantification theory: its justification and realization. *IBM Journal of Research and Development*, 4:28–35, 1960.
- [GK01] A. Goerdt and M. Krivelevich. Efficient recognition of random unsatisfiable k -sat instances by spectral methods. In *18th Symposium on Theoretical Aspects of Computer Science (STACS'01)*, volume 2010 of *Lecture Notes in Computer Science*, pages 294–304. Springer, New York, 2001.
- [GLS93] M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Algorithms and Combinatorics, Vol. 2. Springer-Verlag, Berlin, 1993.
- [Goe96] A. Goerdt. A threshold for unsatisfiability. *Journal of Computer System Science*, 53:469–486, 1996.

- [Gol79] A. Goldberg. On the complexity of the satisfiability problem. In *4th Workshop on Automated Deduction*, pages 1–6. Austin, Texas, 1979.
- [GPB82] A. Goldberg, P. W. Purdom, and C. Brown. Average time analysis of simplified davis-putnam procedures. *Information Processing Letters*, 15:72–75, 1982.
- [Gra99] J. Gramm. Exact algorithms for max2sat and their applications. Diplomarbeit, Universität Tübingen, 1999.
- [Gu92] J. Gu. Efficient local search for very large scale satisfiability problems. *SIGART Bulletin*, 3(1):8–12, 1992.
- [GW93] I. P. Gent and T. Walsh. An empirical analysis of search in gsat. *Journal of Artificial Intelligence Research*, 1:47–59, 1993.
- [GW95] M. X. Goemans and D. P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the Association for Computing Machinery*, 42(6):1115–1145, 1995.
- [GZ03] J. F. Groote and H. Zantema. Resolution and binary decision diagrams cannot simulate each other polynomially. *Discrete Applied Mathematics*, 130(2):157–171, 2003.
- [Hak85] A. Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.
- [Hås01] Johan Håstad. Some optimal inapproximability results. *Journal of the Association for Computing Machinery*, 48:798–859, 2001.
- [HBBM97] Y. Hong, P. A. Beerel, J. R. Burch, and K. L. McMillan. Safe bdd minimization using don't cares. In *34th ACM/IEEE Design Automation Conference*, pages 208–213. IEEE Computer Society Press, Los Alamitos, CA, 1997.
- [HD66] C. Herlinus and C. Dasyppodius. *Analyseis Geometricae Sex Liborum Euclidis*. J. Rihel, Strausbourg, 1566.
- [HD04] J. Huang and A. Darwiche. Toward good elimination orders for symbolic sat solving. In *16th IEEE International Conference on Tools with Artificial Intelligence*, pages 566–573. IEEE Computer Society Press, Los Alamitos, CA, 2004.
- [Hel] C. Helmberg. Sdp. Available from <http://www-user.tu-chemnitz.de/~helmberg/semidef.html>.
- [Hem94] E. Hemaspaandra. Dichotomy theorems for alternation-bound quantified boolean formulas. Technical Report cs.CC/0406006, ACM Computing Research Repository, 1994.
- [Hen49] L. Henkin. The completeness of the first-order functional calculus. *Journal of Symbolic Logic*, 14:159–166, 1949.
- [HHS84] P. L. Hammer, P. Hansen, and B. Simeone. Roof duality, complementation, and persistency in quadratic 0-1 optimization. *Mathematical Programming*, 28:121–155, 1984.
- [Hir00] E. A. Hirsch. A new algorithm for max-2-sat. In *17th Annual Symposium on Theoretical Aspects of Computer Science (STACS '00)*, volume 1770 of *Lecture Notes in Computer Science*, pages 65–73. Springer-Verlag, Berlin, 2000.

- [HJ90] P. Hansen and B. Jaumard. Algorithms for the maximum satisfiability problem. *Computing*, 44:279–303, 1990.
- [HJP93] P. Hansen, B. Jaumard, and G. Plateau. An extension of nested satisfiability. Technical Report G-93-27, Les Cahiers du GERAD, 1993.
- [Hol75] J. H. Holland. *Adaption in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, MI, 1975.
- [Hoo98] H. H. Hoos. *Stochastic local search - methods, models, applications*. PhD thesis, TU Darmstadt, FB Informatik, Darmstadt, Germany, 1998.
- [Hoo99] H. H. Hoos. On the run-time behavior of stochastic local search algorithms for sat. In *16th National Conference on Artificial Intelligence*, pages 661–666. AAAI Press/The MIT Press, Menlo Park, CA, 1999.
- [Hoo02] H. H. Hoos. A mixture model for the behaviour of sls algorithms for sat. In *18th National Conference on Artificial Intelligence*, pages 661–667. AAAI Press/The MIT Press, Menlo Park, CA, 2002.
- [HR68] P. L. Hammer and S. Rudeanu. *Boolean Methods in Operations Research and Related Areas*. Springer-Verlag, Berlin, 1968.
- [HRR63a] P. L. Hammer, I. Rosenberg, and S. Rudeanu. Application of discrete linear programming to the minimization of boolean functions. *Revue Roumaine de Mathématiques Pures et Appliquées*, 8:459–475, 1963.
- [HRR63b] P. L. Hammer, I. Rosenberg, and S. Rudeanu. On the determination of the minima of pseudo-boolean functions. *Studii si Cercetari Matematice*, 14:359–364, 1963.
- [HS71] P. L. Hammer and E. Shliffer. Applications of pseudo-boolean methods to economic problems. *Theory and Decision*, 1:296–308, 1971.
- [HS03] M. T. Hajiaghayi and G. B. Sorkin. The satisfiability threshold of random 3-sat is at least 3.52. Available: <http://arxiv.org/pdf/math.CO/0310193>, 2003.
- [HS05] H.Ĥ. Hoos and T. Stützle. *Stochastic Local Search*. Elsevier, Amsterdam, 2005.
- [HTH02] F. Hutter, D. A. D. Tompkins, and H. H. Hoos. Scaling and probabilistic smoothing: efficient dynamic local search for sat. In *8th International Conference on Principles and Practice of Constraint Programming (CP 2002)*, volume 2470 of *Lecture Notes in Computer Science*, pages 233–248. Springer-Verlag, Berlin, 2002.
- [HZ01] E. Halperin and U. Zwick. Approximation algorithms for max 4-sat and rounding procedures for semidefinite programs. *Journal of Algorithms*, 40(2):184–211, 2001.
- [IM82] A. Itai and J. Makowsky. On the complexity of herbrand’s theorem. Working paper 243, Department of Computer Science, Israel Institute of Technology, 1982.
- [IT03] K. Iwama and S. Tamaki. Improved upper bounds for 3-sat. In *15th ACM-SIAM Symposium on Discrete Algorithms*, pages 328–

328. Society for Industrial and Applied Mathematics, Philadelphia, 2003.
- [ITA⁺06] A. Ishtaiwi, J. R. Thornton, Anbulagan, A. Sattar, and D. N. Pham. Adaptive clause weight redistribution. In *12th International Conference on the Principles and Practice of Constraint Programming (CP 2006)*, volume 4204 of *Lecture Notes in Computer Science*, pages 229–243. Springer, New York, 2006.
- [Iwa89] K. Iwama. Cnf satisfiability test by counting and polynomial average time. *SIAM Journal on Computing*, 18:385–391, 1989.
- [JB06] T. Jussila and A. Biere. Compressing bmc encodings with qbf. In *44th International Workshop on Bounded Model Checking*, number 3 in *Electronic Notes in Theoretical Computer Science* **174**, pages 27–39. Elsevier, the Netherlands, 2006.
- [Jev70] W. S. Jevons. On the mechanical performance of logic inference. *Philosophical Transactions of the Royal Society of London*, 160:497, 1870.
- [JM01] J. Jacob and A. Mishchenko. Unate decomposition of boolean functions. 10th International Workshop on Logic and Synthesis, 2001.
- [Joh74] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and Systems Sciences*, 9:256–278, 1974.
- [JS05] H. S. Jin and F. Somenzi. Circus: A hybrid satisfiability solver. In *7th International Conference on the Theory and Applications of Satisfiability Testing*, volume 3542 of *Lecture Notes in Computer Science*, pages 211–223. Springer, New York, 2005.
- [JW90] R. G. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.
- [Kar72] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computation*, pages 85–104. Plenum Press, New York, 1972.
- [KB99] H. Kleine Büning. An upper bound for minimal resolution refutations. In *12th Workshop on Computer Science Logic (CSL'98)*, volume 1584 of *Lecture Notes in Computer Science*, pages 171–178. Springer-Verlag, Berlin, 1999.
- [KB00] H. Kleine Büning. On subclasses of minimal unsatisfiable formulas. *Discrete Applied Mathematics*, 107(1-3):83–98, 2000.
- [KBKF95] H. Kleine Büning, M. Karpinski, and A. Flögel. Resolution for quantified boolean formulas. *Information and Computation*, 117(1):12–18, 1995.
- [KBL99] H. Kleine Büning and T. Lettmann. *Propositional Logic: Deduction and Algorithms*. Cambridge University Press, 1999.
- [KBSZ07] H. Kleine Büning, K. Subramani, and X. Zhao. Boolean functions as models for quantified boolean formulas. *Journal of Automated Reasoning*, 39(1):49–75, 2007.
- [KGV83] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

- [KH69] R. Kowalski and P. J. Hayes. Semantic trees in automatic theorem-proving. *Machine Intelligence*, 4:87–101, 1969.
- [KKBF91] M. Karpinski, H. Kleine Büning, and A. Flögel. Subclasses of quantified boolean formulas. In *Computer Science Logic (CSL'90)*, volume 533 of *Lecture Notes in Computer Science*, pages 145–155. Springer-Verlag, Berlin, 1991.
- [KKBS88] M. Karpinski, H. Kleine Büning, and P. Schmitt. On the computational complexity of quantified horn clauses. In *Computer Science Logic (CSL'88)*, volume 329 of *Lecture Notes in Computer Science*, pages 129–137. Springer-Verlag, Berlin, 1988.
- [KKK96] L. M. Kirousis, E. Kranakis, and D. Krizanc. A better upper bound for the unsatisfiability threshold. In J. Gu and P. Pardalos, editors, *DIMACS series in Discrete Mathematics and Theoretical Computer Science*, Satisfiability Problem: Theory and Applications, volume 35, pages 643–648. DIMACS, Rutgers, New Jersey, 1996.
- [KKKS98] L. M. Kirousis, E. Kranakis, D. Krizanc, and Y. C. Stamatiou. Approximating the unsatisfiability threshold of random formulae. *Random Structures and Algorithms*, 12:253–269, 1998.
- [KKL04] A. C. Kaporis, L. M. Kirousis, and E. G. Lalas. Selecting complementary pairs of literals. *Electronic Notes in Discrete Mathematics*, 16(1):1–24, 2004.
- [KKL06] A. C. Kaporis, L. M. Kirousis, and E. G. Lalas. The probabilistic analysis of a greedy satisfiability algorithm. *Random Structures and Algorithms*, 28(4):444–480, 2006.
- [Kle43] S. C. Kleene. Recursive predicates and quantifiers. *Transactions of the American Mathematical Society*, 53:41–73, 1943.
- [KMPS95] A. Kamath, R. Motwani, K. Palem, and P. Spirakis. Tail bounds for occupancy and the satisfiability conjecture. *Random Structures and Algorithms*, 7:59–80, 1995.
- [Knu90] D. Knuth. Nested satisfiability. *Acta Informatica*, 28:1–6, 1990.
- [KS94] S. Kirkpatrick and B. Selman. Critical behavior in the satisfiability of random formulas. *Science*, 264:1297–1301, 1994.
- [Kul00] O. Kullmann. Investigations on autark assignments. *Discrete Applied Mathematics*, 107:99–137, 2000.
- [Kul03] O. Kullmann. Lean clause-sets: generalizations of minimally unsatisfiable clause-sets. *Discrete Applied Mathematics*, 130(2):209–249, 2003.
- [KZ97] H. Karloff and U. Zwick. A 7/8-approximation algorithm for max 3sat? In *38th Annual Symposium on the Foundations of Computer Science*, pages 406–415. IEEE Computer Society Press, Los Alamitos, CA, 1997.
- [Las00] J. B. Lasserre. Optimality conditions and lmi relaxations for 0-1 programs. Technical report, LAAS-CNRS, Toulouse, France, 2000.
- [Las02] J. B. Lasserre. An explicit equivalent positive semidefinite program for nonlinear 0-1 programs. *SIAM Journal on Optimization*, 12(3):756–769, 2002.

- [Lau03] M. Laurent. A comparison of the sherali-adams, lovász-schrijver, and lasserre relaxations for 0-1 programming. *Mathematics of Operations Research*, 28(3):470–496, 2003.
- [Lav94] S. Lavine. *Understanding the Infinite*. Harvard University Press, Cambridge, 1994.
- [Lee59] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell Systems Technical Journal*, 38:985–999, 1959.
- [Lew60] C. I. Lewis. *A Survey of Symbolic Logic*. Dover, New York, 1960.
- [Lew78] H. R. Lewis. Renaming a set of clauses as a horn set. *Journal of the Association for Computing Machinery*, 25:134–135, 1978.
- [LH05] C. M. Li and W. Huang. Diversification and determinism in local search for satisfiability. In *Theory and Applications of Satisfiability Testing: 8th International conference (SAT 2005)*, volume 3569 of *Lecture Notes in Computer Science*, pages 158–172. Springer, New York, 2005.
- [Lic82] D. Lichtenstein. Planar formulae and their uses. *SIAM Journal on Computing*, 11:329–343, 1982.
- [Lin65] S. Lin. Computer solutions of the traveling salesman problem. *Bell System Technical Journal*, 44:2245–2269, 1965.
- [LK73] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling salesman problem. *Operations Research*, 21(2):498–516, 1973.
- [Lov79] L. Lovász. On the shannon capacity of a graph. *IEEE Transactions on Information Theory*, 25(1):1–7, 1979.
- [LS91] L. Lovász and A. Schrijver. Cones of matrices and set-functions and 0-1 optimization. *SIAM Journal on Optimization*, 1(2):166–190, 1991.
- [Mar97] J. N. Martin. Aristotle’s natural deduction reconsidered. *History and Philosophy of Logic*, 18:1–15, 1997.
- [MB01] C. B. McDonald and R. E. Bryant. Computing logic-stage delays using circuit simulation and symbolic elmore analysis. In *38 ACM-IEEE Design Automation Conference*, pages 283–288. IEEE Computer Society Press, Los Alamitos, CA, 2001.
- [McC59] E. I. McCluskey, Jr. Minimization of boolean functions. *Bell System Technical Journal*, 35:1417–1444, 1959.
- [McM93] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.
- [Mel92] B. Meltzer. On formally undecidable propositions of principia mathematica and related systems. In *Translation of the German original by Kurt Gödel, 1931*. Basic Books 1962, Reprinted Dover Press, 1992.
- [Min93] S. Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In *30th ACM-IEEE Design Automation Conference*, pages 272–277. IEEE Computer Society Press, Los Alamitos, CA, 1993.
- [Min96] S. Minato. Fast factorization method for implicit cube cover representation. *IEEE Transactions on Computer Aided Design*,

- 15(4):377–384, 1996.
- [Mit97] M. Mitzenmacher. Tight thresholds for the pure literal rule. DEC/SRC Technical Note 1997-011, 1997.
- [MJPL90] S. Minton, M. D. Johnston, A. B. Philips, and P. Laird. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *8th National Conference on Artificial Intelligence*, pages 17–24. AAAI Press/The MIT Press, Menlo Park, CA, 1990.
- [MM02] D. Motter and I. Markov. A compressed breadth-first search for satisfiability. In *4th International Workshop Algorithm on Engineering and Experiments*, volume 2409 of *Lecture Notes in Computer Science*, pages 29–42. Springer, New York, 2002.
- [MMZ⁺01] M. W. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Engineering a (super?) efficient sat solver. In *38th ACM/IEEE Design Automation Conference*, pages 530–535. IEEE Computer Society Press, Los Alamitos, CA, 2001.
- [MR99a] M. Mahajan and V. Raman. Parameterizing above guaranteed values: Maxsat and maxcut. *Journal of Algorithms*, 31:335–354, 1999.
- [MR99b] S. Mahajan and H. Ramesh. Derandomizing approximation algorithms based on semidefinite programming. *SIAM Journal on Computing*, 28(5):1641–1663, 1999.
- [MS72] A. R. Meyer and L. J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *13th Symposium on Switching and Automata Theory*, pages 125–129. IEEE Computer Society Press, Los Alamitos, CA, 1972.
- [MS73] A. R. Meyer and L. J. Stockmeyer. Word problems requiring exponential time. In *5th Symposium on the Theory of Computing*, pages 1–9. Association for Computing Machinery, New York, 1973.
- [MS79] B. Monien and E. Speckenmeyer. 3-satisfiability is testable in $o(1.62^r)$ steps. *Reihe Informatik, Bericht Nr. 3*, Universität-GH Paderborn, 1979.
- [MS85] B. Monien and E. Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics*, 10:287–295, 1985.
- [MSK97] D. McAllester, B. Selman, and H. A. Kautz. Evidence for invariants in local search. In *14th National Conference on Artificial Intelligence*, pages 321–326. AAAI Press/The MIT Press, Menlo Park, CA, 1997.
- [MSL92] D. G. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions for sat problems. In *10th National Conference on Artificial Intelligence*, pages 459–465. AAAI Press/The MIT Press, Menlo Park, CA, 1992.
- [MSS96] J. P. Marques-Silva and K. A. Sakallah. Grasp: a new search algorithm for satisfiability. In *ICCAD'96*, pages 220–227. IEEE Computer Society Press, Los Alamitos, CA, 1996.
- [MZ02] M. Mézard and R. Zecchina. The random k -satisfiability problem: from an analytic solution to an efficient algorithm. *Physical Review*

- E*, 66:056126–056126–27, 2002.
- [MZK⁺99a] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. $2 + p$ -sat: Relation of typical-case complexity to the nature of the phase transition. *Random Structures and Algorithms*, 15(3-4):414–435, 1999.
- [MZK⁺99b] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. Determining computational complexity from characteristic “phase transitions”. *Nature*, 400:133–137, 1999.
- [Nie98] R. Niedermeier. Some prospects for efficient fixed parameter algorithms. In *25th Conference on Current Trends in Theory and Practice of Informatics (SOFSEM '98)*, volume 1521 of *Lecture Notes in Computer Science*, pages 168–185. Springer-Verlag, Berlin, 1998.
- [NR00] R. Niedermeier and P. Rossmanith. New upper bounds for maximum satisfiability. *Journal of Algorithms*, 36:63–88, 2000.
- [NSS57] A. Newell, J. Shaw, and H. Simon. Empirical explorations with the logic theory machine. In *Western Joint Computer Conference*, volume 15, pages 218–239. National Joint Computer Committee, 1957.
- [Pap91] C. H. Papadimitriou. On selecting a satisfying truth assignment. In *32nd IEEE Symposium on Foundations of Computer Science*, pages 163–169. IEEE Computer Society Press, Los Alamitos, CA, 1991.
- [Par03] P. A. Parrilo. Semidefinite programming relaxations for semialgebraic problems. *Mathematical Programming*, 96(2, Series B):293–320, 2003.
- [PB85] P. W. Purdom and C. A. Brown. The pure literal rule and polynomial average time. *SIAM Journal on Computing*, 14:943–953, 1985.
- [PH97] P. W. Purdom and G. N. Haven. Probe order backtracking. *SIAM Journal on Computing*, 26:456–483, 1997.
- [PPZ97] R. Paturi, P. Pudlák, and F. Zane. Satisfiability coding lemma. In *38th IEEE Symposium on Foundations of Computer Science*, pages 566–574. IEEE Computer Society Press, Los Alamitos, CA, 1997.
- [PPZ98] R. Paturi, P. Pudlák, and F. Zane. An improved exponential-time algorithm for k -sat. In *39th IEEE Symposium on Foundations of Computer Science*, pages 628–637. IEEE Computer Society Press, Los Alamitos, CA, 1998.
- [PR75] J. C. Picard and H. D. Ratliff. Minimum cuts and related problems. *Networks*, 5:357–370, 1975.
- [Pra60] D. Prawitz. An improved proof procedure. *Theoria*, 26(2):102–139, 1960.
- [PRF87] J. M. Plotkin, J. W. Rosenthal, and J. Franco. Correction to probabilistic analysis of the davis-putnam procedure for solving the satisfiability problem. *Discrete Applied Mathematics*, 17:295–299, 1987.

- [Pur83] P. W. Purdom. Search rearrangement backtracking and polynomial average time. *Artificial Intelligence*, 21:117–133, 1983.
- [Put93] M. Putinar. Positive polynomials on compact semi-algebraic sets. *Indiana University Mathematics Journal*, 42(3):969–984, 1993.
- [PV05] G. Pan and M. Vardi. Search vs. symbolic techniques in satisfiability solving. In *7th International Conference on the Theory and Applications of Satisfiability Testing*, volume 3542 of *Lecture Notes in Computer Science*, pages 235–250. Springer, New York, 2005.
- [Qui55] W. V. O. Quine. A way to simplify truth functions. *American Mathematical Monthly*, 62:627–631, 1955.
- [Qui59] W. V. O. Quine. On cores and prime implicants of truth functions. *American Mathematical Monthly*, 66:755–760, 1959.
- [Rao71] M. R. Rao. Cluster analysis and mathematical programming. *Journal of the American Statistical Association*, 66:622–626, 1971.
- [Rob63] J. A. Robinson. Theorem-proving on the computer. *Journal of the Association for Computing Machinery*, 10:163–174, 1963.
- [Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12:23–41, 1965.
- [Rob68] J. A. Robinson. The generalized resolution principle. *Machine Intelligence*, 3:77–94, 1968.
- [Ros72] I. G. Rosenberg. Reduction of bivalent maximization to the quadratic case. *Cahiers du Centre d'Etudes de Recherche Operationnelle*, 17:71–74, 1972.
- [Rus02] B. Russell. *Principles of Mathematics, Section 212*. Norton, New York, 1902.
- [SA90] H. D. Sherali and W. P. Adams. A hierarchy of relaxations between the continuous and convex hull representations for zero-one programming problems. *SIAM Journal on Discrete Mathematics*, 3(3):411–430, 1990.
- [SAFS95] J. S. Schlipf, F. Annexstein, J. Franco, and R. Swaminathan. On finding solutions for extended horn formulas. *Information Processing Letters*, 54:133–137, 1995.
- [Sav70] W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and Systems Sciences*, 4:177–192, 1970.
- [Sch73] T. Schaefer. The complexity of satisfiability problems. In *10th ACM Symposium on the Theory of Computing*, pages 1–9. Association for Computing Machinery, New York, 1973.
- [Sch81] H.-P. Schwefel. *Numerical Optimization of Computer Models*. John Wiley & Sons, Chichester, UK, 1981.
- [Sch96] I. Schiermeyer. Pure literal look ahead: a 3-satisfiability algorithm. In J. Franco, G. Gallo, H. Kleine Büning, E. Speckenmeyer, and C. Spera, editors, *1st Workshop on the Theory and Applications of Satisfiability Testing*, Report No.96-230, pages 127–136. Reihe: Angewandte Mathematik und Informatik, Universität zu Köln, 1996.

- [Sch99] U. Schöning. A probabilistic algorithm for k -sat and constraint satisfaction problems. In *40th IEEE Symposium on Foundations of Computer Science*, pages 410–414. IEEE Computer Society Press, Los Alamitos, CA, 1999.
- [Sch02] U. Schöning. A probabilistic algorithm for k -sat based on limited local search and restart. *Algorithmica*, 32:615–623, 2002.
- [Sch05] R. Schuler. An algorithm for the satisfiability problem of formulas in conjunctive normal form. *Journal of Algorithms*, 54(1):40–44, 2005.
- [Scu90] M. G. Scutella. A note on dowling and gallier’s top-down algorithm for propositional horn satisfiability. *Journal of Logic Programming*, 8:265–273, 1990.
- [Sha40] C. E. Shannon. A symbolic analysis of relay and switching circuits. Master’s thesis, Massachusetts Institute of Technology, 1940. Available from <http://hdl.handle.net/1721.1/11173>.
- [SK93] B. Selman and H. A. Kautz. Domain-independent extensions to gsat: solving large structured satisfiability problems. In *13th International Joint Conference on Artificial Intelligence*, pages 290–295. Morgan Kaufmann Publishers, San Francisco, CA, 1993.
- [SKC94] B. Selman, H. A. Kautz, and B. Cohen. Noise strategies for improving local search. In *12th National Conference on Artificial Intelligence*, pages 337–343. AAAI Press/The MIT Press, Menlo Park, CA, 1994.
- [SLM92] B. Selman, H. Levesque, and D. G. Mitchell. A new method for solving hard satisfiability problems. In *10th National Conference on Artificial Intelligence*, pages 440–446. AAAI Press/The MIT Press, Menlo Park, CA, 1992.
- [SM54] E. W. Samson and B. E. Mills. Circuit minimization: algebra and algorithms for new boolean canonical expressions. Technical Report 54-21, Air Force Cambridge Research Center, 1954.
- [SS98] Mary Sheeran and Gunnar Stålmarmark. A tutorial on stålmarmark’s proof procedure for propositional logic. pages 82–99, 1998.
- [SS06] H. M. Sheini and K. A. Sakallah. Pueblo: a hybrid pseudo-boolean sat solver. *Journal on Satisfiability, Boolean Modeling, and Computation*, 2:165–189, 2006.
- [SSH01] D. Schuurmans, F. Southey, and R. C. Holte. The exponentiated subgradient algorithm for heuristic boolean programming. In *17th International Joint Conference on Artificial Intelligence*, pages 334–341. Morgan Kaufmann Publishers, San Francisco, CA, 2001.
- [SSW01] R. Schuler, U. Schöning, and O. Watanabe. An improved randomized algorithm for 3-sat. Technical Report TR-C146, Department of Mathematics and Computer Sciences, Tokyo Institute of Technology, Japan, 2001.
- [Sto36] M. H. Stone. The theory of representation for boolean algebras. *Transactions of the American Mathematical Society*, 40:37–111, 1936.

- [Sto77] L. J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3:1–22, 1977.
- [SW95] R. P. Swaminathan and D. K. Wagner. The arborescence-realization problem. *Discrete Applied Mathematics*, 59:267–283, 1995.
- [SW98] Y. Shang and B. W. Wah. A discrete lagrangian-based global search method for solving satisfiability problems. *Journal of Global Optimization*, 12(1):61–100, 1998.
- [Sze03] S. Szeider. Minimal unsatisfiable formulas with bounded clause-variable difference are fixed-parameter tractable. In *9th International Conference on Computing and Combinatorics (COCOON '03)*, volume 2697 of *Lecture Notes in Computer Science*, pages 548–558. Springer, New York, 2003.
- [Tar44] A. Tarski. Truth and proof. *Philosophy and Phenomenological Research*, 4:341–375, 1944.
- [Tar54] A. Tarski. Contributions to the theory of models. *Indagationes Mathematicae*, 16:572–588, 1954.
- [Tar56] A. Tarski. The concept of truth in formalized languages. In A. Tarski, editor, *Logic Semantics, Metamathematics*. Clarendon Press, Oxford, 1956.
- [Tov84] C. A. Tovey. A simplified NP-complete satisfiability problem. *Discrete Applied Mathematics*, 8:85–89, 1984.
- [Tru94] K. Truemper. Monotone decomposition of matrices. Technical Report UTDCS-1-94, University of Texas at Dallas, 1994.
- [Tru98] K. Truemper. *Effective Logic Computation*. John Wiley & Sons, New York, 1998.
- [Tse68] G. Tseitin. On the complexity of derivation in propositional calculus. In A. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logic, Part 2*, pages 115–125. Consultants Bureau, New York-London, 1968.
- [TTT66] F. Turquette, M. Turquette, and A. Turquette. Peirce's triadic logic. *Transactions of the Charles S. Peirce Society*, 11:71–85, 1966.
- [Tur36] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings, Series 2*, volume 42, pages 230–265. London Mathematical Society, 1936.
- [Urq87] A. Urquhart. Hard examples for resolution. *Journal of the Association for Computing Machinery*, 34:209–219, 1987.
- [US94] T. E. Uribe and M. E. Stickel. Ordered binary decision diagrams and the davis-putnam procedure. In *1st International Conference on Constraints in Computational Logics (CCL '94)*, volume 845 of *Lecture Notes in Computer Science*, pages 34–49. Springer-Verlag, Berlin, 1994.
- [Ven80] J. Venn. On the diagrammatic and mechanical representation of propositions and reasonings. *Dublin Philosophical Magazine and Journal of Science*, 9(59):1–18, 1880.
- [vM99] H. van Maaren. Elliptic approximations of propositional formulae.

- Discrete Applied Mathematics*, 96/97:223–244, 1999.
- [vM00] H. van Maaren. A short note on some tractable classes of the satisfiability problem. *Information and Computation*, 158(2):125–130, 2000.
- [vMvN05] H. van Maaren and L. van Norden. Sums of squares, satisfiability and maximum satisfiability. In *8th International conference on the Theory and Applications of Satisfiability Testing (SAT 2005)*, volume 3569 of *Lecture Notes in Computer Science*, pages 293–307. Springer, New York, 2005.
- [vMvNH08] H. van Maaren, L. van Norden, and M. J. H. Heule. Sums of squares based approximation algorithms for max-sat. *Discrete Applied Mathematics*, 156(10):1754–1779, 2008.
- [WF96] R. Wallace and E. Freuder. Comparative studies of constraint satisfaction and davis-putnam algorithms for maximum satisfiability problems. In D. Johnson and M. Trick, editors, *Cliques, Coloring and Satisfiability*, volume 26 in the DIMACS series, pages 587–615. American Mathematical Society, 1996.
- [Wit33] L. Wittgenstein. *Tractatus Logico-Philosophicus*. Reprinted by K. Paul, Trench, Trubner, London, 1933.
- [Wor95] N. C. Wormald. Differential equations for random processes and random graphs. *Annals of Applied Probability*, 5:1217–1235, 1995.
- [Wra77] C. Wrathall. Complete sets and the polynomial-time hierarchy. *Theoretical Computer Science*, 3:22–33, 1977.
- [WSVe00] H. Wolkowicz, R. Saigal, and L. Vandenberghe (eds.). *Handbook of Semidefinite Programming*. Kluwer Academic Publishers, Boston, MA, 2000.
- [WW99] Z. Wu and B. W. Wah. Trap escaping strategies in discrete lagrangian methods for solving hard satisfiability and maximum satisfiability problems. In *16th National Conference on Artificial Intelligence*, pages 673–678. AAAI Press/The MIT Press, Menlo Park, CA, 1999.
- [WW00] Z. Wu and B. W. Wah. An efficient global-search strategy in discrete lagrangian methods for solving hard satisfiability problems. In *17th National Conference on Artificial Intelligence*, pages 310–315. AAAI Press/The MIT Press, Menlo Park, CA, 2000.
- [XRS02] H. Xu, R. A. Rutenbar, and K. A. Sakallah. sub-sat: A formulation for related boolean satisfiability with applications in routing. In *2002 ACM International Symposium on Physical Design*, pages 182–187. Association for Computing Machinery, New York, 2002.
- [ZSM03] H. Zhang, H. Shen, and F. Manyá. Exact algorithms for max-sat. In *International Workshop on First-order Theorem Proving (FTP 2003)*. 2003.
- [ZZ04] X. Zhao and W. Zhang. An efficient algorithm for maximum boolean satisfiability based on unit propagation, linear programming, and dynamic weighting. Preprint, Department of Computer Science, Washington University, 2004.

Chapter 2

CNF Encodings

Steven Prestwich

2.1. Introduction

Before a combinatorial problem can be solved by SAT methods, it must usually be transformed (*encoded*) to *conjunctive normal form* (CNF): a conjunction of clauses $\bigwedge_i c_i$, each clause c_i being a disjunction of literals $\bigvee_j l_j$, and each literal l_j being either a Boolean variable v or its negation \bar{v} . CNF has the advantage of being a very simple form, leading to easy algorithm implementation and a common file format. Unfortunately there are several ways of encoding most problems and few guidelines on how to choose among them, yet the choice of encoding can be as important as the choice of SAT solver.

This chapter reviews theoretical and empirical work on CNF encodings. In Section 2.2 we describe techniques for transforming from a formula in propositional logic to CNF (and 3-CNF), in particular the Tseitin encodings and some variants, the encoding of extensional and intensional constraints, and the DIMACS file format for CNF. In Section 2.3 we present case studies in CNF encoding, illustrating the techniques of this section and other modelling aspects. In Section 2.4 we discuss what makes one encoding better than another. Finally, Section 2.5 concludes the chapter.

Some of the topics mentioned in this chapter are covered in greater depth elsewhere in this book, but we cover them briefly to aid the discussion. For the sake of readability we shall omit some details from examples of SAT encodings. In particular, instead of $\bigwedge_{j=1}^m (\bigvee_{i=1}^n v_{i,j})$ we shall simply write $\bigvee_i v_{i,j}$ and leave it to the reader to quantify the variable i over its range, and to form conjunctions of clauses by quantifying the free variable j over its range. For full details please consult the various papers cited below.

2.2. Transformation to CNF

Propositional logic formulae can be transformed to CNF in several ways, but this may lose a great deal of structural information. Some of this information can be computed from the CNF encoding and can be used to boost the performance of both DPLL [Li00, OGMS02] (the Davis-Putnam-Logemann-Loveland

[DP60, DLL62] family of backtracking-based SAT algorithms) and local search [Seb94], but some of it is intractable to compute [LM98]. To avoid this difficulty, solvers for non-CNF problems have been devised [AG93, VG88, KMS97, MS06, Ott97, PTS07, Sta01, TBW04]. These techniques can yield great improvements on certain structured problems.

However, CNF currently remains the norm. The reason might be that tried and tested heuristics, based on problem features such as clause lengths and the number of occurrences of literals, would need to be adapted to non-CNF formulae. In this section we describe methods for transforming propositional logic formulae and constraints into CNF.

2.2.1. Transformation by Boolean algebra

A propositional logic formula can be transformed to a logically equivalent CNF formula by using the rules of Boolean algebra. For example consider the propositional formula

$$(a \rightarrow (c \wedge d)) \vee (b \rightarrow (c \wedge e))$$

(taken from [TBW04]). The implications can be decomposed:

$$((a \rightarrow c) \wedge (a \rightarrow d)) \vee ((b \rightarrow c) \wedge (b \rightarrow e))$$

The conjunctions and disjunctions can be rearranged:

$$\begin{aligned} & ((a \rightarrow c) \vee (b \rightarrow c)) \wedge ((a \rightarrow c) \vee (b \rightarrow e)) \wedge \\ & ((a \rightarrow d) \vee (b \rightarrow c)) \wedge ((a \rightarrow d) \vee (b \rightarrow e)) \end{aligned}$$

The implications can be rewritten as disjunctions, and duplicated literals removed:

$$(\bar{a} \vee \bar{b} \vee c) \wedge (\bar{a} \vee \bar{b} \vee c \vee e) \wedge (\bar{a} \vee \bar{b} \vee c \vee d) \wedge (\bar{a} \vee \bar{b} \vee d \vee e)$$

Finally, subsumed clauses can be removed, leaving the conjunction

$$(\bar{a} \vee \bar{b} \vee c) \wedge (\bar{a} \vee \bar{b} \vee d \vee e)$$

This example reduces to a compact CNF formula, but in general this method yields exponentially large formulae.

2.2.2. Transformation by Tseitin encoding

Conversion to CNF is more usually achieved using the well-known Tseitin encodings [Tse83], which generate a linear number of clauses at the cost of introducing a linear number of new variables, and generate an equisatisfiable formula (satisfiable if and only if the original formula is satisfiable). Tseitin encodings work by adding new variables to the CNF formula, one for every subformula of the original formula, along with clauses to capture the relationships between these new variables and the subformulae. On the above example the Tseitin encoding would introduce a variable f_1 with definition

$$f_1 \leftrightarrow (c \wedge d)$$

to represent subformula $(c \wedge d)$. This definition can be reduced to clausal form:

$$(\overline{f_1} \vee c) \wedge (\overline{f_1} \vee d) \wedge (\overline{c} \vee \overline{d} \vee f_1)$$

Similarly it would introduce a definition

$$f_2 \leftrightarrow (c \wedge e)$$

which reduces to clauses

$$(\overline{f_2} \vee c) \wedge (\overline{f_2} \vee e) \wedge (\overline{c} \vee \overline{e} \vee f_2)$$

Applying these definitions the original formula is reduced to

$$(\overline{a} \vee f_1) \wedge (\overline{b} \vee f_2)$$

Next two more variables would be introduced with definitions

$$f_3 \leftrightarrow (\overline{a} \vee f_1)$$

and

$$f_4 \leftrightarrow (\overline{b} \vee f_2)$$

and clauses

$$(\overline{f_3} \vee \overline{a} \vee f_1) \wedge (a \vee f_3) \wedge (\overline{f_1} \vee f_3) \wedge (\overline{f_4} \vee \overline{b} \vee f_2) \wedge (b \vee f_4) \wedge (\overline{f_2} \vee f_4)$$

The formula is now reduced to

$$(f_3 \vee f_4)$$

Finally, a variable would be introduced with definition

$$f_5 \leftrightarrow (f_3 \vee f_4)$$

and clauses

$$(\overline{f_5} \vee f_3 \vee f_4) \wedge (\overline{f_3} \vee f_5) \wedge (\overline{f_4} \vee f_5)$$

Tseitin CNF encodings are linear in the size of the original formula as long as the Boolean operators that appear in the formula have linear clausal encodings. The operators *and*, *or*, *not*, *nand*, *nor* and *implies* all have linear clausal encodings. In practice we might choose not to define some variables, for example $(f_3 \vee f_4)$ is already in clausal form so we need not define f_5 . We might also choose to expand some non-clausal subformulae.

Alternatives to the Tseitin encodings have been described, mainly in the context of Boolean circuits, but the ideas are relevant to SAT in general. Consider the following example (taken from [TBW04]). Suppose we have a formula $X \vee (p \wedge q)$ where X is a large unsatisfiable formula and p, q are Boolean variables. A Tseitin encoding will include auxiliary variables x, y, z defined by $x \leftrightarrow X$, $y \leftrightarrow (p \wedge q)$ and $z \leftrightarrow (x \vee y)$. Now suppose that DPLL first selects variables y, z and sets them to true. Then unit propagation will set p, q to true. Next suppose that it selects variable x and sets it to false. Then DPLL only needs to find any variable assignment that falsifies X (which should be easy) and the problem is solved. However, if x was instead set to true then DPLL might be forced to perform a lengthy search to refute X , before backtracking to set x to false. Under the assignments $[\overline{x}, y, z]$ the X variables have become *don't care variables*: their values do not affect the satisfiability of the problem and they have become *unobservable*. A survey of transformation techniques for avoiding this source of inefficiency is given in [Vel05].

2.2.3. Transformation from CNF to 3-CNF

3-CNF formulae have exactly three literals in each clause. These problems are interesting in their own right, and random 3-CNF problems have been extensively studied. However, any SAT problem can be transformed into 3-CNF. SAT is known to be NP-complete, so this transformation proves the completeness of 3-CNF, by showing that 3-CNF is as expressive as CNF.

A well-known transformation is as follows. Suppose we have a SAT instance with clauses c_1, \dots, c_m over variables u_1, \dots, u_n , where each clause c_i may contain any positive number of literals. This can be transformed to 3-CNF as follows. For a clause $c_i = (z_1 \vee \dots \vee z_k)$ there are four cases:

- ($k = 1$) Create two new variables $y_{i,1}, y_{i,2}$ and replace c_i by the clauses $(z_1 \vee y_{i,1} \vee y_{i,2})$, $(z_1 \vee \overline{y_{i,1}} \vee y_{i,2})$, $(z_1 \vee y_{i,1} \vee \overline{y_{i,2}})$ and $(z_1 \vee \overline{y_{i,1}} \vee \overline{y_{i,2}})$.
- ($k = 2$) Create one new variable $y_{i,1}$ and replace c_i by the clauses $(z_1 \vee z_2 \vee y_{i,1})$ and $(z_1 \vee z_2 \vee \overline{y_{i,1}})$.
- ($k = 3$) Leave clause c_i unchanged.
- ($k > 3$) Create new variables $y_{i,1}, y_{i,2}, \dots, y_{i,k-3}$ and replace c_i by the clauses $(z_1 \vee z_2 \vee y_{i,1})$, $(\overline{y_{i,1}} \vee z_3 \vee y_{i,2})$, $(\overline{y_{i,2}} \vee z_4 \vee y_{i,3})$, \dots , $(\overline{y_{i,k-3}} \vee z_{k-1} \vee z_k)$.

This transformation does not seem to have many practical applications, but it may be useful for generating 3-CNF benchmarks, or as a preprocessor for a specialised algorithm that only accepts 3-CNF problems.

2.2.4. Extensional constraints in CNF

One way of modelling a problem as SAT is first to model it as a constraint satisfaction problem (CSP), then to apply a standard encoding from CSP to SAT. A finite-domain CSP has variables $v_1 \dots v_n$ each with a finite domain $\text{dom}(v_i)$ of values, and constraints prescribing prohibited combinations of values (alternatively, constraints may prescribe *allowed* combinations of assignments). The problem is to find an assignment of values to all variables such that no constraint is violated. An example of a constraint is $(v_3 = 2, v_4 = 2)$ which prohibits the combination of assignments $v_3 = 2$ and $v_4 = 2$. This is a *binary* constraint because it contains only two variables, and a binary CSP contains only binary constraints.

The most natural and widely-used encoding from CSP to SAT is the *direct encoding* (also called the *sparse encoding*). A SAT variable $x_{v,i}$ is defined as true if and only if the CSP variable v has the domain value i assigned to it. The direct encoding consists of three sets of clauses. Each CSP variable must take at least one domain value, expressed by *at-least-one* clauses:

$$\bigvee_i x_{v,i}$$

No CSP variable can take more than one domain value, expressed by *at-most-one* clauses:

$$\overline{x_{v,i}} \vee \overline{x_{v,j}}$$

Conflicts are enumerated by *conflict* clauses:

$$\overline{x_{v,i}} \vee \overline{x_{w,j}}$$

An alternative to the direct encoding is the *support encoding* (defined only for binary CSPs) [Gen02, Kas90] in which conflict clauses are replaced by *support* clauses. Suppose that $S_{v,j,w}$ are the *supporting values* in $\text{dom}(v)$ for value $j \in \text{dom}(w)$: these are the values in $\text{dom}(w)$ that do not conflict with $w = j$. Then add a support clause

$$\overline{x_{w,j}} \vee \left(\bigvee_{i \in S_{v,j,w}} x_{v,i} \right)$$

A third alternative is the *log encoding*, first defined for Hamiltonian circuit, clique and colouring problems [IM94] and since used for other problems including planning [EMW97] and vertex colouring [VG07]. Its motivation is to exponentially reduce the number of SAT variables with respect to the direct encoding. Each CSP variable/domain value bit has a corresponding SAT variable, and each conflict has a corresponding clause prohibiting that combination of bits in the specified CSP variables. Define variables $x_{b,v}$ where $x_{b,v} = 1$ if and only if bit b of the domain value assigned to v is 1 (here we number the bits from 0 with the 0^{th} bit being least significant). For example to prohibit the combination $[p = 2, q = 1]$ where p, q have domains $\{0, 1, 2\}$ and therefore require two bits each, we must prohibit the combination $[\overline{x_{0,p}}, x_{1,p}, \overline{x_{0,q}}, x_{1,q}]$ by adding a conflict clause $(x_{0,p} \vee \overline{x_{1,p}} \vee x_{0,q} \vee \overline{x_{1,q}})$. No at-least-one or at-most-one clauses are required, but if the domain size of v is not a power of 2 then we must add clauses to prohibit combinations of bits representing non-domain values. For example here we must prevent the bit pattern $[x_{0,p}, x_{1,p}]$ representing $p = 3$ by adding a clause $\overline{x_{0,p}} \vee \overline{x_{1,p}}$, and similarly for q .

To illustrate the three encodings we use a simple graph colouring problem as an example, with two adjacent vertices v and w and three available colours $\{0, 1, 2\}$. Each vertex must take exactly one colour, but because they are adjacent they cannot take the same colour. This problem has six solutions: $[v = 0, w = 1]$, $[v = 0, w = 2]$, $[v = 1, w = 0]$, $[v = 1, w = 2]$, $[v = 2, w = 0]$ and $[v = 2, w = 1]$. The direct encoding of the problem contains at-least-one clauses

$$x_{v,0} \vee x_{v,1} \vee x_{v,2} \quad x_{w,0} \vee x_{w,1} \vee x_{w,2}$$

at-most-one clauses

$$\begin{array}{ccc} \overline{x_{v,0}} \vee \overline{x_{v,1}} & \overline{x_{v,0}} \vee \overline{x_{v,2}} & \overline{x_{v,1}} \vee \overline{x_{v,2}} \\ \overline{x_{w,0}} \vee \overline{x_{w,1}} & \overline{x_{w,0}} \vee \overline{x_{w,2}} & \overline{x_{w,1}} \vee \overline{x_{w,2}} \end{array}$$

and conflict clauses

$$\overline{x_{v,0}} \vee \overline{x_{w,0}} \quad \overline{x_{v,1}} \vee \overline{x_{w,1}} \quad \overline{x_{v,2}} \vee \overline{x_{w,2}}$$

The support encoding contains the same at-least-one and at-most-one clauses, plus support clauses

$$\begin{array}{ccc} x_{v,1} \vee x_{v,2} \vee \overline{x_{w,0}} & x_{v,0} \vee x_{v,2} \vee \overline{x_{w,1}} & x_{v,0} \vee x_{v,1} \vee \overline{x_{w,2}} \\ x_{w,1} \vee x_{w,2} \vee \overline{x_{v,0}} & x_{w,0} \vee x_{w,2} \vee \overline{x_{v,1}} & x_{w,0} \vee x_{w,1} \vee \overline{x_{v,2}} \end{array}$$

The log encoding contains only conflict clauses

$$\begin{array}{c} x_{v,0} \vee x_{v,1} \vee x_{w,0} \vee x_{w,1} \\ \overline{x_{v,0}} \vee \overline{x_{v,1}} \vee \overline{x_{w,0}} \vee \overline{x_{w,1}} \\ x_{w,0} \vee \overline{x_{w,1}} \vee x_{w,0} \vee \overline{x_{w,1}} \end{array}$$

and *prohibited value clauses* to forbid non-domain values

$$\overline{x_{v,0}} \vee \overline{x_{v,1}} \qquad \overline{x_{w,0}} \vee \overline{x_{w,1}}$$

There are also further encodings. *Hierarchical encodings* recursively subdivide domains [Vel07], and *representative-sparse encoding* [BHN14] are based on hierarchical and direct encodings. A hybrid of the log and support encodings called the *log support encoding* was devised by [Gav07]. Variables and prohibited value clauses as in the log encoding, and some of the conflict clauses are replaced by support clauses. A variant based on Gray codes was also proposed. The *binary transform* of [FP01] is a variant of the log encoding that avoids the use of prohibited value clauses. The direct encoding without at-most-one clauses is often used with SAT local search algorithms, and has been called the *multivalued encoding* [Pre04]. SAT solutions under this encoding represent *bundles* of solutions to the CSP, formed by taking the Cartesian product of the CSP domain values allowed in the SAT solution, and to extract a single solution we simply select any allowed value for each CSP variable. For binary CSPs the *maximal encoding* [Pre04] eliminates all but the maximal bundles (in which no additional domain values are allowed) by adding *maximality clauses* to the multivalued encoding:

$$x_{v,i} \vee \bigvee_{w,j} x_{w,j}$$

where w, j range over the assignments to other variables $w = j$ that conflict with $v = i$. The maximal encoding breaks a form of dominance between CSP solutions and can speed up all-solution search. On the above example the maximality clauses are:

$$x_{v,0} \vee x_{w,0} \qquad x_{v,1} \vee x_{w,1} \qquad x_{v,2} \vee x_{w,2}$$

For CSPs with integer linear constraints, a better alternative is the *order encoding* [AM04, CB94, TTKB09]. In this encoding a SAT variable $v_{x,a}$ represents a *primitive comparison* $x \leq a$ for integer variable x and upper bound a . Other comparisons (linear constraints) are transformed into primitive comparisons via a method described in [TTKB09], illustrated here by an example. For integer variables x, y, z each with domain $\{0, 1, 2, 3\}$ the comparison $x + y < z - 1$ becomes

$$\begin{array}{lll} v_{x,-1} \vee v_{y,-1} \vee \overline{v_{z,1}} & v_{x,-1} \vee v_{y,0} \vee \overline{v_{z,2}} & v_{x,-1} \vee v_{y,1} \vee \overline{v_{z,3}} \\ v_{x,0} \vee v_{y,-1} \vee \overline{v_{z,2}} & v_{x,0} \vee v_{y,0} \vee \overline{v_{z,3}} & v_{x,1} \vee v_{y,-1} \vee \overline{v_{z,3}} \end{array}$$

(Variables corresponding to non-existent domain values such as -1 can be eliminated.) The order relation is represented by *axiom clauses* of the form

$$\overline{v_{x,a}} \vee v_{x,a+1}$$

The order encoding was proved to transform a tractable CSP into a tractable SAT problem by [PJ11]. Improved variants were developed by [AS14] to handle problems with large domains, incorporating techniques from the log and support encodings, and Multi and Binary Decision Diagrams. The *compact order encoding* [TTB11] combines ideas from the order and log encodings, and the *representative-order encoding* [Ngu17] uses the order encoding in partitions of the variables, which are represented by selected variables.

2.2.5. Intensional constraints in CNF

Not all constraints are conveniently expressed extensionally by listing conflicts, and we may wish to SAT-encode constraints that are usually expressed intentionally. For example the *at-most-one* constraint states that at most one of a set of variables $x_1 \dots x_n$ can be true. We shall use this example to illustrate alternative encodings of an intensional constraint. The often-used *pairwise encoding* treats the constraint extensionally, and simply enumerates $O(n^2)$ at-most-one clauses:

$$\overline{x_i} \vee \overline{x_j}$$

where $i < j$. The *ladder encoding* described by [GN04] is more compact: define new variables $y_1 \dots y_{n-1}$ and add *ladder validity clauses*

$$\overline{y_{i+1}} \vee y_i$$

and *channelling clauses* expanded from

$$x_i \leftrightarrow (y_{i-1} \wedge \overline{y_i})$$

The ladder adds $O(n)$ new variables but only $O(n)$ clauses. The *binary encoding* was proposed by [FPDN05] and later by [Pre07a] who named it the *bitwise encoding* (we use the former name here). Define new Boolean variables b_k where $k = 1 \dots \lceil \log_2 n \rceil$. Add clauses

$$\overline{x_i} \vee b_k \quad [\text{or } \overline{b_k}]$$

if bit k of $i - 1$ is 1 [or 0]. This encoding has $O(\log n)$ new variables and $O(n \log n)$ binary clauses: more literals but fewer new variables than the ladder encoding. The pairwise encoding is acceptable for fairly small n , while for large n the ladder encoding gives good results with DPLL, and the binary encoding gives good results with local search algorithms. A *relaxed ladder encoding* defined in [Pre07a] also gave improved performance with local search. Further at-most-one encodings are surveyed in [NM15] and we briefly mention them here. In the *commander encoding* [KK07] the SAT variables are partitioned into subsets, each represented by a *commander variable* to reduce the number of clauses. The *product encoding* [Che10] is recursive and based on Cartesian products, and requires few clauses and variables. The *bimander encoding* [NM15] is based on the commander encoding but replaces the commander variables by variables from the binary encoding.

A useful generalisation of the at-most-one constraint is the *cardinality constraint*, which states that at most (or at least) k of a set of variables $x_1 \dots x_n$ can be true. This can be compactly SAT-encoded and a survey of such encodings was given by [MSL07], who also show that SAT solver performance on such encodings can be improved by effectively ignoring the auxiliary variables that they introduce. [BB03a] used a similar arithmetic-based method to that of [War98] for linear inequalities, but with a unary representation of integer variables to improve propagation. [Sin05] described two improved encodings: one smaller and based on a sequential counter, the other with better propagation and based on a parallel counter. [ANORC09, ANORC11b] based their encoding on sorting networks.

[CZI10] based a more compact encoding on a different type of sorting network. [OLH⁺13] used a different type of arithmetic network to obtain another compact encoding, which performed well on MaxSAT problems.

Integer linear inequalities (or *pseudo-Boolean constraints*) further generalise cardinality constraints and are of the form:

$$\sum_i w_i x_i \leq k$$

for some integer weights w_i (interpreting false as 0 and true as 1); the \leq relation may be replaced by $<$, \geq , $>$ or $=$. These constraints can be handled either directly by generalising SAT to linear pseudo-Boolean problems, or indirectly by finding SAT encodings. [War98] SAT-encoded hardware arithmetic circuits to model these constraints. [ES06] described three encoding methods based on Binary Decision Diagrams, arithmetic networks and sorting networks. [ANORC11a] used Binary Decision Diagrams to obtain an encoding with improved propagation.

Another intensional constraint is the *parity* constraint $(\bigoplus_{i=1}^n v_i) \leftrightarrow p$ which states that p is true if and only if there is an even number of true variables in the set $\{v_1, \dots, v_n\}$. This can be encoded simply by enumerating all possible combinations of v_i truth values, together with their parity p , but this creates exponentially many clauses and is only reasonable for small constraints. A more practical linear method due to [Li00] decomposes the constraint by introducing new variables f_j :

$$(v_1 \oplus f_1) \leftrightarrow p \quad (v_2 \oplus f_2) \leftrightarrow f_1 \quad \dots \quad (v_{n-3} \oplus f_{n-3}) \leftrightarrow f_{n-2} \quad v_n \leftrightarrow f_{n-1}$$

These binary and ternary constraints are then put into clausal form by enumerating cases, for example $(v_1 \oplus f_1) \leftrightarrow p$ becomes:

$$(\overline{v_1} \vee \overline{f_1} \vee p) \wedge (v_1 \vee \overline{f_1} \vee \overline{p}) \wedge (\overline{v_1} \vee f_1 \vee \overline{p}) \wedge (v_1 \vee f_1 \vee p)$$

A commander encoding-style hybrid of the enumeration and linear methods was used in [Pre07b] to improve local search performance.

2.2.6. The DIMACS file format for CNF

A file format for SAT problems in CNF (as well as vertex colouring and clique problems) was devised in the DIMACS Challenge of 1993 [JT96], and has been followed ever since. (A format for general SAT was also proposed but does not appear to have been generally adopted.) Having a common file format has facilitated the collection of SAT benchmark problems in the SATLIB web site ¹, the Beijing challenges ² and the regular SAT solver competitions ³, which have stimulated a great deal of research into efficient algorithms and implementations. This is similar to the use of AMPL files for MIP problems, and file formats have also been introduced into Constraint Programming for solver competitions.

The format is as follows. At the start is a *preamble* containing information about the file. These optional comment lines begin with the letter “c”:

¹<http://www.cs.ubc.ca/~hoos/SATLIB/>

²<http://www.cirl.uoregon.edu/crawford/beijing>

³<http://www.satcompetition.org/>

c This is an example of a comment line.

Then a line of the form

p cnf variables clauses

states the number of Boolean variables and clauses in the file, where *variables* and *clauses* are positive integers and variables are numbered $1 \dots variables$. The rest of the file contains the clauses. Each clause is represented by a list of non-zero integers, followed by a zero which represents the end of the clause. The integers must be separated by spaces, tabs or newlines. A positive integer i represents a positive literal with variable number i , while a negative integer $-i$ represents a negative literal with variable number i . For example the line

1 5 -8 0

represents the clause $v_1 \vee v_5 \vee \overline{v_8}$. Clauses may run over more than one line, spaces and tabs may be inserted freely between integers, and both the order of literals in a clause and the order of clauses in the file are irrelevant.

2.3. Case studies

In this section we discuss concrete SAT models. We use simple problems to illustrate CSP-to-SAT encodings, alternative and non-obvious variable definitions, alternative combinations of clauses, the selective application of Tseitin encodings, the exploitation of subformula polarity, the use of implied and symmetry-breaking clauses, special modelling techniques for local search, and some common sources of error.

2.3.1. N-queens

We begin with an example of CSP-to-SAT modelling: the well-known n-queens problem. Consider a generalised chess board, which is a square divided into $n \times n$ smaller squares. The problem is to place n queens on it in such a way that no queen attacks any other. A queen *attacks* another if it is on the same row, column or diagonal (in which case both attack each other). How many ways are there to model this problem as a SAT problem? One or two models might spring to mind, but in a classic paper [Nad90] Nadel presents no fewer than nine constraint models (denoted by Q1–Q9), some representing families of models and one being isomorphic to a SAT problem. There are several ways of SAT-encoding a CSP, so we immediately have a rather large number of ways of modelling n-queens as SAT. We now discuss some of these ways, plus additional variants.

Nadel's model Q1 is perhaps the most obvious one. For each row define a variable with a finite domain of n values, denoting the column in which the queen on that row is placed. Add binary constraints to prevent attacks, by forbidding pairs of variables from taking values that place queens in the same column (the values are identical) or cause queens to attack along a diagonal (their absolute difference matches the absolute difference between the variable numbers). Row attacks are implicitly forbidden because each row variable takes exactly one value.

Model Q2 is very similar to Q1, but variables correspond to columns and values to rows. A direct encoding of Q1 or Q2 gives almost identical results, except for clauses ensuring that each row (Q1) or column (Q2) contains at least one queen. A further option is to use both sets of clauses, because in any n -queens solution there is a queen in each row and column. This is an example of adding implied clauses to a model: an implied clause is one that is implied by other clauses in the model, and therefore logically redundant. Implied clauses have been shown to speed up both backtracking [ABC⁺02] and local search [CI96, KD95].

Model Q6 is already a SAT model. For each board square (i, j) define a Boolean variable $v_{i,j}$, which is true if and only if a queen is placed on it. To prevent partially-filled boards (including the empty board) from being solutions add clauses

$$\bigvee_i v_{i,j}$$

to ensure that each row contains a queen; alternatively we could ensure that each column contains a queen; or both. To prevent attacks add a clause

$$(\overline{v_{i,j}} \vee \overline{v_{i',j'}})$$

for each distinct pair of squares $(i, j), (i', j')$ that are on the same row, column or diagonal and therefore attack each other. It is worth pointing out here that there is no point in adding multiple versions of the same clause, for example:

$$(\overline{v_{2,5}} \vee \overline{v_{3,6}}) \quad (\overline{v_{3,6}} \vee \overline{v_{2,5}})$$

This would occur if we only considered distinct attacking squares: $(i, j) \neq (i', j')$. Instead the attacking squares used in conflict clauses should satisfy $(i, j) \prec (i', j')$ for some total ordering \prec on squares; for example the lexicographical ordering $(i, j) \prec (i', j')$ if and only if (i) $i < i'$ or (ii) $i = i'$ and $j < j'$. This type of needless duplication is easy to commit when SAT-encoding. Model Q6 is almost identical to the direct encoding of Q1 or Q2, modulo details of whether we ensure no more than one queen per row, column or both.

Model Q5 is quite different. It associates a variable with each queen, and each variable has a domain ranging over the set of squares on the board. A direct encoding would define a Boolean variable $v_{q,i,j}$ for each queen q and each square (i, j) , which is true if and only if that queen is placed on that square. Add clauses to prevent two queens from being placed on the same square:

$$(\overline{v_{q,i,j}} \vee \overline{v_{q',i,j}})$$

for all pairs q, q' such that $q < q'$ (to avoid generating two equivalent versions of each clause). Add clauses to prevent attacks:

$$(\overline{v_{q,i,j}} \vee \overline{v_{q',i',j'}})$$

for each pair of queens $q < q'$ and each distinct pair of attacking squares $(i, j) \neq (i', j')$. Alternatively we could enumerate each distinct pair of queens $q \neq q'$ and each pair of attacking squares such that $(i, j) \prec (i', j')$. However, it would be incorrect to enforce both $q < q'$ and $(i, j) \prec (i', j')$ because this would (for

example) prevent the attacking configuration of queen 1 at (1,1) and queen 2 at (2,2), but it would not prevent the attacking configuration of queen 2 at (1,1) and queen 1 at (2,2). Great care must be taken with these issues when SAT-encoding.

A possible drawback with Q5 is that it contains a great deal of symmetry: every solution has $n!$ SAT models because the queens can be permuted. To eliminate this symmetry we can add further clauses, for example by forcing queens to be placed in squares with increasing indices:

$$(\overline{v_{q,i,j}} \vee \overline{v_{q',i',j'}})$$

where $q > q'$ and $(i, j) < (i', j')$. N-queens itself also has 8 symmetries: any solution can be reflected 2 times and rotated 4 times. To break these symmetries and thus reduce the size of the search space we may add further clauses, but we shall not consider these details.

Nor shall we consider Nadel's other constraint models, two of which define a variable for each upward or downward diagonal instead of each row or column: these again lead to SAT-encodings similar to Q6. But there are other ways of encoding the attack constraints. For example, suppose we define the Boolean variables $v_{i,j}$ of Q6. We must ensure that at most one queen is placed on any given row, column or diagonal, which in Q6 was achieved by enumerating all possible pairs of attacking queens. This is the pairwise encoding of the at-most-one constraint, but instead we could use a more compact encoding such as the ladder encoding. Another possibility is to model support instead of conflict in model Q1, replacing several conflict clauses

$$\overline{v_{i,j}} \vee \overline{v_{i',j'}}$$

by a clause

$$\overline{v_{i,j}} \vee \left(\bigvee_{j' \in C_{i,j,i'}} v_{i,j'} \right)$$

where $C_{i,j,i'}$ is the set of columns whose squares on row i' are not attacked by a queen at (i, j) . This clause states that if a queen is placed on square (i, j) then the queen on row j' must be placed on a non-attacking column: note that its correctness depends on clauses ensuring that exactly one queen can be placed in any row. This model corresponds to a support encoding of Q1. We could also exchange the roles of rows and columns to obtain a support encoding of Q2, or use conflict or support clauses from Q1 and Q2 to obtain hybrid models.

The n-queens problem illustrates the approach of first modelling a problem as a CSP, then SAT-encoding the CSP. It shows that, even for a very simple problem, there may be a choice of problem features to encode as variables. Even after making this fundamental choice there may be alternative sets of clauses giving a logically correct model. It provides examples of implied clauses and symmetry breaking clauses. It also illustrates that care must be taken to avoid accidentally excluding solutions by adding too many clauses.

2.3.2. All-interval series

For our next example we take the all-interval series (AIS) problem, an arithmetic problem first used by Hoos [Hoo98] to evaluate local search algorithms, and in-

spired by a problem occurring in serial musical composition. There is a constructive solution to the AIS problem that requires no search so it is inherently easy. However, the easily-found solution is highly regular and other solutions might be more interesting to musicians. The problem is to find a permutation of the numbers $\{0, \dots, n - 1\}$ such that the set of differences between consecutive numbers contains no repeats. For example, a solution for $n = 11$ (taken from [GMS03]) with differences written underneath the numbers is:

$$\begin{array}{cccccccccccc} 0 & 10 & 1 & 9 & 2 & 8 & 3 & 7 & 4 & 6 & 5 \\ & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \end{array}$$

First define Boolean variables $s_{i,j}$ each of which is true if and only if the i^{th} number is the integer j . AIS is a sequence of integers so each number takes exactly one value:

$$\left(\bigvee_i s_{i,j} \right) \wedge (\overline{s_{i,j}} \vee \overline{s_{i,j'}})$$

We may model differences as follows: if the formula

$$\bigvee_j (s_{i,j} \wedge s_{i+1,j \pm k})$$

is true then the difference between numbers i and $i + 1$ is k , so we can enforce distinct differences by the formula:

$$\overline{\bigvee_j (s_{i,j} \wedge s_{i+1,j \pm k})} \vee \overline{\bigvee_j (s_{i',j} \wedge s_{i'+1,j \pm k})}$$

for distinct i, i' and relevant k values. (The meaning of $j \pm k$ here is that clauses should be generated using both $j + k$ and $j - k$, wherever the resulting numbers are within the range of the second index of the s variable.) A CNF encoding of this formula may be obtained by expansion, giving a set of quaternary clauses of the form

$$\overline{s_{i_1,j_1}} \vee \overline{s_{i_2,j_2}} \vee \overline{s_{i_3,j_3}} \vee \overline{s_{i_4,j_4}}$$

Alternatively, we may introduce Tseitin variable definitions

$$d_{i,k} \leftrightarrow \left[\bigvee_j (s_{i,j} \wedge s_{i+1,j \pm k}) \right]$$

which means that $d_{i,k}$ is true if and only if the difference between numbers i and $i + 1$ is k . We then replace the difference formulae by clauses

$$\overline{d_{i,k}} \vee \overline{d_{i',k}}$$

and CNF-encode the $d_{i,k}$ definitions. However, we can be more compact than the Tseitin method here. It was noted in [PG86] that only clauses corresponding to positive [negative] implications are logically necessary when transforming a subformula with negative [positive] *polarity*. The polarity of a subformula is

positive [negative] if it only occurs under an even [odd] number of negations. In this case each subformula

$$\bigvee_j (s_{i,j} \wedge s_{i+1,j \pm k})$$

only occurs negatively, so we only need clauses derived from the implication

$$d_{i,k} \leftarrow \left[\bigvee_j (s_{i,j} \wedge s_{i+1,j \pm k}) \right]$$

yielding clauses of the form

$$\overline{s_{i,j}} \vee \overline{s_{i+1,j \pm k}} \vee d_{i,k}$$

This is convenient, as the \rightarrow implication would require either a larger expansion or more auxiliary variables.

To this encoding may be added several families of implied clauses. The $s_{i,j}$ variables encode a permutation of the integers $0 \dots n$ so each integer occurs at most once:

$$\overline{s_{i,j}} \vee \overline{s_{i',j}}$$

The d variables encode a sequence of differences:

$$\bigvee_i d_{i,k}$$

and

$$\overline{d_{i,j}} \vee \overline{d_{i,j'}}$$

These differences form a permutation of the numbers $1 \dots n$:

$$\overline{d_{i,j}} \vee \overline{d_{i',j}}$$

We have now arrived at Hoos's original AIS encoding [Hoo98]. A further study of AIS encodings was presented by [ABC⁺02]. They experimented with omitting the at-most-one clauses

$$\overline{s_{i,j}} \vee \overline{s_{i,j'}}$$

and

$$\overline{d_{i,k}} \vee \overline{d_{i,k'}}$$

which is safe because of the clauses ensuring that the two sets of variables encode permutations. Thus they treated the at-most-one clauses as implied clauses, and also introduced a less-obvious set of implied clauses:

$$\overline{d_{i,k}} \vee s_{i-1,m} \wedge \overline{d_{i,k}} \vee s_{i,m}$$

where $k > n/2$ (n is assumed to be even) and $n - k \leq m \leq k - 1$. In experiments with local search algorithms, different combinations of these implied clauses had either very good or very bad effects on performance, showing the unpredictable computational effects of implied clauses.

As pointed out by Gent et al. [GMS03] the AIS problem contains symmetry, which can be broken by adding further clauses. An AIS sequence may be reversed to give another sequence, so we may add clauses

$$\overline{s_{0,j}} \vee \overline{s_{1,j'}}$$

where $j' < j$ forcing the second number to be greater than the first. We may also subtract each value from $n - 1$ to obtain another sequence so we can add clauses to ensure that the first value is no greater than $n/2$:

$$\overline{s_{0,j}}$$

where $j \leq n/2$. Gent et al. also use a clever technique to boost the performance of a constraint solver by a factor of 50. They point out a *conditional symmetry* of AIS, which is a symmetry that occurs during search when a subset of the variables have been assigned values. Breaking the conditional symmetry is not easy in the original AIS model, but they generalise AIS to a new problem containing additional symmetry (rotation of sequence values) that can easily be broken, which incidentally also breaks the symmetry and conditional symmetry in AIS.

The AIS example illustrates the unpredictable but potentially beneficial effects of adding implied clauses, the selective use of Tseitin variables, the exploitation of subformula polarity to reduce the size of the encoding, the use of non-obvious implied clauses, and the possibility of better models that might be found by ingenuity.

2.3.3. Stable marriages

In the above examples there was nothing very surprising about the variable definitions themselves: they mostly corresponded to assignments of constraint variables to domain values. In our next example, taken from a study by Gent et al. [GIM⁺01], the variable definitions are less obvious. In the stable marriage problem we have n men and n women. Each man ranks the women in order of preference, and women similarly rank men. The problem is to marry men and women in a *stable* way, meaning that there is no incentive for any two individuals to elope and marry each other. In the version of the problem considered, incomplete preference lists are allowed, so that a man [woman] might be unacceptable to some women [men]. A person is willing to leave his/her current partner for a new partner only if he/she is either unmatched or considers the new partner better than the current partner. A pair who mutually prefer each other are a blocking pair, and a matching without blocking pairs is stable. The problem of finding a stable matching is not NP-complete but an extended version allowing preference ties is, and can be modelled in a similar way; here we discuss only the simpler problem.

A direct encoding of Gent et al.'s first constraint model would contain a Boolean variable for each man-woman pair, which is true when they are matched. However, they also present a second model that is more compact and already in SAT form. They define a variable $x_{i,p}$ to be true if and only if man i is either unmatched or matched to the woman in position p or later in his preference

list, where $1 \leq p \leq L_{m,i}$ and $L_{m,i}$ is the length of his list. They also define $x_{i,L_{m,i}+1}$ which is true if and only if man i is unmatched. Similarly they define variables $y_{j,q}$ for each woman j . Note that each SAT variable corresponds to a set of possibilities, unlike those in our previous examples; also that marriages are modelled indirectly via the preference lists.

The clauses are as follows. Each man or woman is either matched with someone in their preference list or is unmatched, which is enforced by the unit clauses

$$x_{i,1} \wedge y_{j,1}$$

If a man gets his p^{th} choice then he certainly gets his $p+1^{\text{th}}$ choice:

$$\overline{x_{i,p}} \vee x_{i,p+1}$$

Similarly for women:

$$\overline{y_{j,q}} \vee y_{j,q+1}$$

The x and y variables must be linked so that if man i is matched to woman j then woman j is also matched to man i . We can express the fact that man i is matched to the woman in position p in his preference list indirectly by the assignments $[x_{i,p}, \overline{x_{i,p+1}}]$ so the clauses are:

$$(\overline{x_{i,p}} \vee x_{i,p+1} \vee y_{j,q}) \wedge (\overline{x_{i,p}} \vee x_{i,p+1} \vee \overline{y_{j,q+1}})$$

where p is the rank of woman j in the preference list of man i , and q is the rank of man i in the preference list of woman j . Similarly:

$$(\overline{y_{j,q}} \vee y_{j,q+1} \vee x_{i,p}) \wedge (\overline{y_{j,q}} \vee y_{j,q+1} \vee \overline{x_{i,p+1}})$$

Stability is enforced by clauses stating that if man i is matched to a woman he ranks below woman j then woman j must be matched to a man she ranks no lower than man i :

$$\overline{x_{i,p}} \vee \overline{y_{j,q+1}}$$

where the man finds the woman acceptable (mentioned in the preference list). Similarly:

$$\overline{y_{j,q}} \vee \overline{x_{i,p+1}}$$

This problem provides a nice illustration of the use of non-obvious variable definitions.

Here it is worth pointing out another common source of error in SAT encoding: omitting common-sense axioms. The meaning of variable $x_{i,p}$ being true is that man i is matched to the woman in position p or later in his list. This must of course be explicitly stated in the SAT encoding via clauses $\overline{x_{i,p}} \vee x_{i,p+1}$. In the same way, if we are trying to construct a tree, a DAG or some other data structure then we must include axioms for those structures. However, in some cases these axioms are implied by other clauses. For example, if we are trying to construct a permutation of n numbers with some property then we would normally use at-least-one and at-most-one clauses, because the permutation is a list of length n with each position containing exactly one number. But if we add clauses stating that no two positions contain the same number then the at-most-one clauses become redundant.

2.3.4. Modelling for local search

So far we have largely ignored an important aspect of SAT modelling: the search algorithm that will be applied to the model. It is not immediately clear that the choice of algorithm should affect the choice of model, but it has been found that local search sometimes requires different models than DPPL if it is to be used effectively. This may partly explain the relatively poor performance of local search algorithms on industrial problems in the regular SAT solver competitions: they were perhaps modelled with DPPL in mind. [Pre02] showed that two particularly hard problems for local search can be solved by remodelling. Here we briefly summarise the SAT modeling techniques that were used.

A problem that makes heavy use of parity constraints is *minimal disagreement parity learning* described in [CKS94]. DPPL (and extended versions) on standard encodings is much more successful than local search on this problem. The explanation seems to be that standard encodings use a similar structure to the linear encoding of the parity constraint described in Section 2.2.5, which is very compact but creates a long chain of variable dependencies. These are known to slow down local search [KMS97], and when they form long chains they may cause local search to take polynomial or even exponential time to propagate truth assignments [Pre02, WS02]. A key to solving large parity learning instances by local search turns out to be a different encoding of the parity constraint [Pre02]. Decompose a large parity constraint $\bigoplus_{i=1}^n v_i = p$ into $\beta + 1$ smaller ones:

$$\bigoplus_{i=1}^{\alpha} v_i \equiv p_1 \quad \bigoplus_{i=\alpha+1}^{2\alpha} v_i \equiv p_2 \quad \dots \quad \bigoplus_{i=n-\alpha+1}^n v_i \equiv p_{\beta} \quad \text{and} \quad \bigoplus_{i=1}^{\beta} p_i \equiv p$$

where p_1, \dots, p_{β} are new variables and $n = \alpha \times \beta$ (in experiments $\beta \approx 10$ gave the best results). Now expand these small parity constraints into clauses. This exponentially increases the number of clauses in the model yet greatly boosts local search performance, allowing a standard local search algorithm to solve the notorious 32-bit instances of [CKS94]. This was previously achieved only by using an enhanced local search algorithm that exploited an analysis of the problem [PTS07]. Note that a similar result could be obtained by eliminating some of the variables in the linear encoding, thus a promising modelling technique for local search is to collapse chains of dependency by selective variable elimination.

Another hard problem for local search is the well-known Towers of Hanoi expressed as a STRIPS planning problem (see elsewhere in this book for background on modelling planning problems as SAT). Towers of Hanoi consists of P pegs (or towers) and D disks of different sizes; usually $P = 3$. All D disks are initially on the first peg. A solution is a sequence of moves that transfers all disks to the third peg with the help of the second peg so that (i) only one disk can be moved at a time; (ii) only the disk on top can be moved; (iii) no disk can be put onto a smaller disk. There always exists a plan with $2^D - 1$ steps. No SAT local search algorithm previously solved the problems with 5 and 6 disks in a reasonable time and 4 disks required a specially-designed algorithm, while DPPL algorithms can solve the 6-disk problem. However, 6 disks can be solved quite quickly by local search after remodelling the problem [Pre02]. Firstly, a more compact STRIPS

model was designed, which allowed a standard local search algorithm to solve 4 disks (it also improved DPLL performance and is not relevant here). Secondly, parallel plans were allowed, but not enforced, so that instead of a single serial plan there are a choice of many plans with varying degrees of parallelism. This increases the solution density of the SAT problem and allowed local search to solve 5 disks. Thirdly, implied clauses were added to logically connect variables that were otherwise linked only by chains of dependency. This is another way of reducing the effects of such chains on local search, and was first described in [WS02] for an artificial SAT problem. In planning-as-SAT the state at time t is directly related to the state at time $t + 1$ but not to states at more distant times, via (for example) frame axioms which in *explanatory* form are:

$$\tau_{p,t} \wedge \overline{\tau_{p,t+1}} \rightarrow \left(\bigvee_a \alpha_{a,t} \right)$$

where a ranges over the set of actions with delete effect p . This axiom states that if property p is true at time t but false at time $t + 1$ then at least one action must have occurred at time t to delete property p . (Similar axioms are added for properties that are true at time t but false at time $t + 1$.) These axioms create chains of dependency between the τ variables, which can be “short-circuited” by a generalisation of the frame axioms that connect states at arbitrary times:

$$\tau_{pt} \wedge \overline{\tau_{p,t'}} \rightarrow \left[\bigvee_{t'=t}^{t'-1} \left(\bigvee_a \alpha_{a,t'} \right) \right]$$

where $t' > t$. Interestingly, and somewhat mysteriously, in both applications optimal performance was obtained by adding only a small, randomly-chosen subset of these *long-range dependencies*.

In summary, two encoding properties that seem to make a problem hard for local search are: (i) low solution density, which can sometimes be artificially increased by remodelling the problem; and (ii) the presence of chains of dependent variables, which can either be (a) collapsed by variable elimination or (b) short-circuited by adding long-range dependencies. However, these techniques have an erratic and sometimes very bad effect on DPLL performance [Pre02].

2.4. Desirable properties of CNF encodings

What features of a CNF encoding make it better than another? In this section we discuss features that have been proposed as desirable.

2.4.1. Encoding size

By the *size* of an encoding we may mean the number of its clauses, literals or variables. Firstly, consider the number of clauses. As noted above, this can be reduced by introducing new variables as in the Tseitin encodings, possibly transforming an intractably large encoding into a manageable one. However,

the resulting encoding is not always the best in practice. For example [BM00] describes an encoding of a round robin tournament scheduling problem with n teams that has $O(n^6)$ clauses, which gave much better results than a considerably more compact encoding with auxiliary variables and only $O(n^4)$ clauses. Moreover, adding implied clauses [ABC⁺02, CI96, KD95], symmetry breaking clauses [CGLR96] and blocked clauses [Kul99] can greatly improve performance.

Secondly, the total number of literals in an encoding, computed by summing the clause lengths, also has a strong effect on runtime overheads (though this is reduced by the implementation technique of watched literals). As a measure of the memory used by an encoding it is more accurate than the number of clauses. However, the same examples show that minimising this measure of size does not necessarily give the best results. Moreover, the CSP support encoding has given superior results to the direct encoding, despite typically containing more literals.

Thirdly, consider the number of variables in an encoding. Several papers have analysed the worst- or average-case runtime of SAT algorithms in terms of the number of variables in the formula (for example [Sch99]), so it seems worthwhile to minimise the number of variables. However, a counter-example is the log encoding and its variants, which generally give poor performance both with DPLL [FP01, VG07] and local search algorithms [EMW97, Hoo98]. There may also be modelling reasons for not minimising the number of variables: Gent & Lynce [GL05] describe a SAT-encoding of the Social Golfer problem that contains several times more variables than necessary, which was found to be more convenient for expressing the problem constraints.

In practice, reducing the size of an encoding is no guarantee of performance, no matter how it is measured. Nevertheless, small encodings are worth aiming for because computational results can be very unpredictable, and all else being equal a smaller encoding is preferable.

2.4.2. Consistency properties

DPLL uses unit propagation extensively, and it is interesting to ask what type of consistency reasoning is achieved by unit propagation on a given encoding. The support encoding [Gen02, Kas90] for binary CSPs has the property that unit propagation maintains arc consistency, whereas the direct encoding maintains a weaker form of consistency called forward checking [Wal00], and the log encoding maintains a weaker form still [Wal00]. The encodings of [BHW03] generalise the support encoding to higher forms of consistency, and those of [Bac07] achieve global arc consistency for arbitrary constraints. The cardinality constraint SAT-encoding of [BB03b] has the property that unit propagation maintains generalised arc consistency. Implementing consistency in this way is much easier than directly implementing filtering algorithms, especially if we want to interleave them or combine them with techniques such as backjumping and learning [Bac07].

The strength of consistency achieved by unit propagation is reflected in DPLL performance on many problems. Despite the indirect nature of this implementation technique, in some cases it even outperforms specialised constraint solvers [QW07]. However, it is well-known in Constraint Programming that stronger consistency is not always worth achieving because of its additional overheads, and

[BHW03] report similar findings with some of their encodings. The same is true of implied clauses, symmetry breaking clauses and blocked clauses: adding them can greatly boost the power of unit propagation, but can also have unpredictable effects on performance (see [Pre03b] and the example discussed in Section 2.3.2).

2.4.3. Solution density

Another interesting property of an encoding is its solution density, which can be defined as the number of solutions divided by 2^n where n is the number of SAT variables. The multivalued CSP encoding mentioned in Section 2.2.4 increases solution density and is often used with local search. The idea that higher solution density makes a problem easier to solve seems natural, and is usually assumed to have at least some effect on search performance [CFG⁺96, CB94, HHI03, Par97, SGS00, Yok97]. It was pointed out by Clark et al. [CFG⁺96] that one might expect a search algorithm to solve a (satisfiable) problem more quickly if it has higher solution density. In detailed experiments on random problems across solvability phase transitions they did find a correlation between local and backtrack search performance and the number of solutions, though not a simple one. Problem features other than solution density also seem to affect search cost, for example Yokoo [Yok97] showed that adding more constraints to take a random problem beyond the phase transition removes local minima, making them easier for local search algorithms to solve despite removing solutions.

An obvious counter-example to the notion that higher density problems are easier is the CSP log encoding [IM94]. This has only a logarithmic number of variables in the CSP domain sizes, therefore a much higher SAT solution density than the direct encoding for any given CSP. Yet it generally performs poorly with both DPLL and local search algorithms. However, the binary transform of [FP01] is of similar form to the log encoding but with even higher solution density, and gives better results with local search (at least in terms of local search moves) [Pre03a]. It is possible to modify the direct encoding to increase its solution density, but this gives very erratic results [Pre03a].

2.4.4. Summary

It seems that no single feature of an encoding can be used to characterise how good it is. To further confuse the issue, it turns out that a good encoding for one algorithm might be a poor encoding for another algorithm. This was found with the modified models for parity learning and Towers of Hanoi in Section 2.3.4: standard encodings of these problems are hard for local search but can be solved by DPLL, while modified encodings boost local search but can have drastic effects on DPLL performance. Similarly, in Section 2.2.5 different encodings of the at-most-one constraint were found to be better for DPLL and local search. There is also evidence that adding clauses to break symmetry makes a problem harder to solve by local search but easier by DPLL [Pre03b]. Finally, new variables introduced by Tseitin encoding are *dependent* on others, and as mentioned in Section 2.3.4 these are known to slow down local search [KMS97, Pre02, WS02]. Thus the standard Tseitin approach might be inappropriate when combined with local search. However, it does not necessarily harm DPLL performance, because

the selection of these variables for branching can be delayed until unit propagation eliminates them — though this assumes a measure of control over the branching heuristic.

2.5. Conclusion

There are usually many ways to model a given problem in CNF, and few guidelines are known for choosing among them. There is often a choice of problem features to model as variables, and some might take considerable thought to discover. Tseitin encodings are compact and mechanisable but in practice do not always lead to the best model, and some subformulae might be better expanded. Some clauses may be omitted by polarity considerations, and implied, symmetry breaking or blocked clauses may be added. Different encodings may have different advantages and disadvantages such as size or solution density, and what is an advantage for one SAT solver might be a disadvantage for another. In short, CNF modelling is an art and we must often proceed by intuition and experimentation. But enumerating known techniques and results, as we have done in this chapter, helps to clarify the alternatives.

References

- [ABC⁺02] T. Alsinet, R. Béjar, A. Cabiscol, C. Fernández, and F. Manyà. Minimal and redundant sat encodings for the all-interval-series problem. In *Fifth Catalanian Conference on Artificial Intelligence*, pages 139–144. Springer, 2002. Lecture Notes in Computer Science vol. 2504.
- [AG93] A. Armando and E. Giunchiglia. Embedding complex decision procedures inside an interactive theorem prover. *Ann. Math. Artif. Intell.*, 8(3-4):475–502, 1993.
- [AM04] C. Ansótegui and F. Manyà. Mapping problems with finite-domain variables into problems with boolean variables. In *Seventh International Conference on Theory and Applications of Satisfiability Testing*, pages 1–15. Springer, 2004. Lecture Notes in Computer Science vol. 3542.
- [ANORC09] R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. Cardinality networks and their applications. In *Twelfth International Conference on Theory and Applications of Satisfiability Testing*, pages 167–180. Springer, 2009. Lecture Notes in Computer Science vol. 5584.
- [ANORC11a] I. Abío, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. Bdds for pseudo-boolean constraints - revisited. In *Fourteenth International Conference on Theory and Applications of Satisfiability Testing*, pages 61–75. Springer, 2011. Lecture Notes in Computer Science vol. 6695.
- [ANORC11b] R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. Cardinality networks: a theoretical and empirical study. *Constraints*, 16(2):195–221, 2011.

- [AS14] I. Abío and P. J. Stuckey. Encoding linear constraints into sat. In *Twentieth International Conference on Principles and Practice of Constraint Programming*, pages 75–91. Springer, 2014. Lecture Notes in Computer Science vol. 8656.
- [Bac07] F. Bacchus. Gac via unit propagation. In *Thirteenth International Conference on Principles and Practice of Constraint Programming*, pages 133–147. Springer, 2007. Lecture Notes in Computer Science vol. 4741.
- [BB03a] O. Bailleux and Y. Bouffhad. Efficient cnf encoding of boolean cardinality constraints. In *Ninth International Conference on Principles and Practice of Constraint Programming*, pages 108–122. Springer, 2003. Lecture Notes in Computer Science vol. 2833.
- [BB03b] O. Bailleux and Y. Bouffhad. Efficient cnf encoding of boolean cardinality constraints. In *Ninth International Conference on Principles and Practice of Constraint Programming*, pages 108–122. Springer, 2003. Lecture Notes in Computer Science vol. 2833.
- [BHN14] P. Barahona, S. Hölldobler, and V.-H. Nguyen. Representative encodings to translate finite csps into sat. In *Eleventh International Conference on Integration of AI and OR Techniques in Constraint Programming*, pages 251–267. Springer, 2014. Lecture Notes in Computer Science vol. 8451.
- [BHW03] C. Bessière, E. Hebrard, and T. Walsh. Local consistencies in sat. In *Sixth International Conference on Theory and Applications of Satisfiability Testing*, pages 299–314, 2003. Lecture Notes in Computer Science vol. 2919.
- [BM00] R. Béjar and F. Manyà. Solving the round robin problem using propositional logic. In *Seventeenth National Conference on Artificial Intelligence*, pages 262–266, 2000. Austin, Texas.
- [CB94] J. M. Crawford and A. B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Twelfth National Conference on Artificial Intelligence*, pages 1092–1097. AAAI Press, 1994. vol. 2.
- [CFG+96] D. Clark, J. Frank, I. Gent, E. MacIntyre, N. Tomov, and T. Walsh. Local search and the number of solutions. In *Second International Conference on Principles and Practice of Constraint Programming*, pages 119–133, 1996.
- [CGLR96] J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *International Conference on Principles of Knowledge Representation and Reasoning*, pages 148–159, 1996.
- [Che10] J.-C. Chen. A new sat encoding of the at-most-one constraint. In *Tenth International Workshop of Constraint Modelling and Reformulation*, 2010.
- [CI96] B. Cha and K. Iwama. Adding new clauses for faster local search. In *Fourteenth National Conference on Artificial Intelligence*, pages 332–337, 1996. American Association for Artificial Intelligence.
- [CKS94] J. M. Crawford, M. J. Kearns, and R. E. Shapire. The minimal

- disagreement parity problem as a hard satisfiability problem. Technical report, Computational Intelligence Research Laboratory and AT&T Bell Labs, 1994.
- [CZI10] M. Codish and M. Zazon-Ivry. Pairwise cardinality networks. In *Sixteenth international conference on Logic for programming, artificial intelligence, and reasoning*, pages 154–172. Springer, 2010. Lecture Notes in Computer Science vol. 6355.
- [DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.
- [DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association of Computing Machinery*, 7(3), 1960.
- [EMW97] M. Ernst, T. Millstein, and D. Weld. Automatic sat-compilation of planning problems. In *Fifteenth International Joint Conference on Artificial Intelligence*, pages 1169–1176, 1997. Nagoya, Japan.
- [ES06] N. Eén and N. Sörensson. Translating pseudo-boolean constraints into sat. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006.
- [FP01] A. Frisch and T. Peugniez. Solving non-boolean satisfiability problems with stochastic local search. In *Seventeenth International Joint Conference on Artificial Intelligence*, 2001. Seattle, Washington.
- [FPDN05] A. M. Frisch, T. J. Peugniez, A. J. Doggett, and P. W. Nightingale. Solving non-boolean satisfiability problems with stochastic local search: A comparison of encodings. *J. Autom. Reasoning*, 35:143–179, 2005.
- [Gav07] M. Gavarelli. The log-support encoding of csp into sat. In *Thirteenth International Conference on Principles and Practice of Constraint Programming*, pages 815–822. Springer, 2007. Lecture Notes in Computer Science vol. 4741.
- [Gen02] I. P. Gent. Arc consistency in sat. In *Fifteenth European Conference on Artificial Intelligence*, pages 121–125. IOS Press, 2002.
- [GIM⁺01] I. P. Gent, R. W. Irving, D. Manlove, P. Prosser, and B. M. Smith. A constraint programming approach to the stable marriage problem. In *Seventh International Conference on Principles and Practice of Constraint Programming*, pages 225–239. Springer-Verlag, 2001. Lecture Notes In Computer Science vol. 2239.
- [GL05] I. P. Gent and I. Lynce. A sat encoding for the social golfer problem. In *Workshop on Modelling and Solving Problems with Constraints*, 2005. IJCAI’05.
- [GMS03] I. P. Gent, I. McDonald, and B. M. Smith. Conditional symmetry in the all-interval series problem. In *Third International Workshop on Symmetry in Constraint Satisfaction Problems*, 2003.
- [GN04] I. P. Gent and P. Nightingale. A new encoding of alldifferent into sat. In *Third International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, 2004. CP’04.
- [HHI03] Y. Hanatani, T. Horiyama, and K. Iwama. Density condensation of

- boolean formulas. In *Sixth International Conference on the Theory and Applications of Satisfiability Testing*, pages 126–133. Springer, 2003. Lecture Notes In Computer Science vol. 2919.
- [Hoo98] H. H. Hoos. *Stochastic Local Search — Methods, Models, Applications*. PhD thesis, Darmstadt University of Technology, 1998.
- [IM94] K. Iwama and S. Miyazaki. Sat-variable complexity of hard combinatorial problems. In *IFIP World Computer Congress*, pages 253–258. Elsevier Science B. V., North-Holland, 1994.
- [JT96] D. S. Johnson and M. A. Trick, editors. *Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge*, volume 26. American Mathematical Society, 1996.
- [Kas90] S. Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence*, 45:275–286, 1990.
- [KD95] K. Kask and R. Dechter. Gsat and local consistency. In *Fourteenth International Joint Conference on Artificial Intelligence*, pages 616–622. Morgan Kaufmann, 1995.
- [KK07] W. Klieber and G. Kwon. Efficient cnf encoding for selecting 1 from n objects. In *Fourth Workshop on Constraint in Formal Verification*, 2007.
- [KMS97] H. Kautz, D. McAllester, and B. Selman. Exploiting variable dependency in local search. In *Poster Sessions of the Fifteenth International Joint Conference on Artificial Intelligence*, 1997.
- [Kul99] O. Kullmann. New methods for 3-sat decision and worst-case analysis. *Theoretical Computer Science*, 223(1-2):1–72, 1999.
- [Li00] C. M. Li. Integrating equivalence reasoning into davis-putnam procedure. In *Seventeenth National Conference on Artificial Intelligence*, pages 291–296, 2000.
- [LM98] J. Lang and P. Marquis. Complexity results for independence and definability in propositional logic. In *International Conference on Principles of Knowledge Representation and Reasoning*, pages 356–367, 1998.
- [MS06] R. Muhammad and P. J. Stuckey. A stochastic non-cnf sat solver. In *Trends in Artificial Intelligence, Ninth Pacific Rim International Conference on Artificial Intelligence*, pages 120–129. Springer, 2006. Lecture Notes in Computer Science vol. 4099.
- [MSL07] J. Marques-Silva and I. Lynce. Towards robust cnf encodings of cardinality constraints. In *Thirteenth International Conference on Principles and Practice of Constraint Programming*, pages 483–497. Springer, 2007. Lecture Notes in Computer Science vol. 4741.
- [Nad90] B. A. Nadel. Representation selection for constraint satisfaction: a case study using n-queens. *IEEE Expert: Intelligent Systems and Their Applications*, 5(3):16–23, 1990.
- [Ngu17] V.-H. Nguyen. Sat encodings of finite-csp domains: A survey. In *Eighth International Symposium on Information and Communication Technology*, pages 84–91. ACM, 2017.
- [NM15] V.-H. Nguyen and S. T. Mai. A new method to encode the at-

- most-one constraint into sat. In *Sixth International Symposium on Information and Communication Technology*, pages 46–53. ACM, 2015.
- [OGMS02] R. Ostrowski, É. Grégoire, B. Mazure, and L. Saïs. Recovering and exploiting structural knowledge from cnf formulas. In *Eighth International Conference on Principles and Practice of Constraint Programming*, pages 185–199. Springer-Verlag, 2002. Lecture Notes in Computer Science vol. 2470.
- [OLH⁺13] T. Ogawa, Y. Liu, R. Hasegawa, M. Koshimura, and H. Fujita. Modulo based cnf encoding of cardinality constraints and its application to maxsat solvers. In *International Conference on Tools with Artificial Intelligence*, pages 9–17. IEEE, 2013.
- [Ott97] J. Otten. On the advantage of a non-clausal davis-putnam procedure. Technical report, Technische Hochschule Darmstadt, Germany, 1997.
- [Par97] A. Parkes. Clustering at the phase transition. In *Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference*, pages 340–345. AAAI Press / MIT Press, 1997.
- [PG86] D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986.
- [PJ11] J. Petke and P. Jeavons. The order encoding: From tractable csp to tractable sat. In *Fourteenth International Conference on Theory and Applications of Satisfiability Testing*, pages 371–372. Springer, 2011. Lecture Notes in Computer Science vol. 6695.
- [Pre02] S. D. Prestwich. Sat problems with chains of dependent variables. *Discrete Applied Mathematics*, 3037:1–22, 2002.
- [Pre03a] S. D. Prestwich. Local search on sat-encoded colouring problems. In *Sixth International Conference on the Theory and Applications of Satisfiability Testing*, pages 105–119. Springer, 2003. Lecture Notes in Computer Science vol. 2919.
- [Pre03b] S. D. Prestwich. Negative effects of modeling techniques on search performance. *Annals of Operations Research*, 118:137–150, 2003.
- [Pre04] S. D. Prestwich. Full dynamic substitutability by sat encoding. In *Tenth International Conference on Principles and Practice of Constraint Programming*, pages 512–526. Springer, 2004. Lecture Notes in Computer Science vol. 3258.
- [Pre07a] S. D. Prestwich. Finding large cliques using sat local search. In F. Benhamou, N. Jussien, and B. O’Sullivan, editors, *Trends in Constraint Programming*, pages 269–274. ISTE, 2007. Chapter 15.
- [Pre07b] S. D. Prestwich. Variable dependency in local search: Prevention is better than cure. In *Tenth International Conference on Theory and Applications of Satisfiability Testing*, pages 107–120. Springer, 2007. Lecture Notes in Computer Science vol. 4501.
- [PTS07] D. N. Pham, J. R. Thornton, and A. Sattar. Building structure into local search for sat. In *Twentieth International Joint Conference on*

- Artificial Intelligence*, pages 2359–2364, 2007. Hyderabad, India.
- [QW07] C.-G. Quimper and T. Walsh. Decomposing global grammar constraints. In *Thirteenth International Conference on Principles and Practice of Constraint Programming*, pages 590–604. Springer, 2007. Lecture Notes in Computer Science vol. 4741.
- [Sch99] U. Schöning. A probabilistic algorithm for k-sat and constraint satisfaction problems. In *Fortieth Annual Symposium on Foundations of Computer Science*, pages 410–414. IEEE Computer Society, 1999.
- [Seb94] R. Sebastiani. Applying gsat to non-clausal formulas. *Journal of Artificial Intelligence Research*, 1:309–314, 1994. research note.
- [SGS00] J. Singer, I. P. Gent, and A. Smaill. Backbone fragility and the local search cost peak. *Journal of Artificial Intelligence Research*, 12:235–270, 2000.
- [Sin05] C. Sinz. Towards an optimal cnf encoding of boolean cardinality constraints. In *Eleventh International Conference on Principles and Practice of Constraint Programming*, pages 827–831. Springer, 2005. Lecture Notes in Computer Science vol. 3709.
- [Sta01] Z. Stachniak. Non-clausal reasoning with definite theories. *Fundam. Inform.*, 48(4):363–388, 2001.
- [TBW04] C. Thiffault, F. Bacchus, and T. Walsh. Solving non-clausal formulas with dpll search. In *Seventh International Conference on Theory and Applications of Satisfiability Testing*, pages 147–156. Springer, 2004. Lecture Notes in Computer Science vol. 3542.
- [Tse83] G. Tseitin. On the complexity of derivation in propositional calculus. *Automation of Reasoning: Classical Papers in Computational Logic*, 2:466–483, 1983. Springer-Verlag.
- [TTB11] T. Tanjo, N. Tamura, and M. Banbara. A compact and efficient sat-encoding of finite domain csp. In *Fourteenth International Conference on Theory and Applications of Satisfiability Testing*, pages 375–376. Springer, 2011. Lecture Notes in Computer Science vol. 6695.
- [TTKB09] N. Tamura, A. Taga, S. Kitagawa, and M. Banbara. Compiling finite linear csp into sat. *Constraints*, 14(2):254–272, 2009.
- [Vel05] M. N. Velev. Comparison of schemes for encoding unobservability in translation to sat. In *Asia and South Pacific Design Automation Conference*, pages 1056–1059, 2005.
- [Vel07] M. N. Velev. Exploiting hierarchy and structure to efficiently solve graph coloring as sat. In *International Conference on Computer-Aided Design*, pages 135–142. IEEE/ACM, 2007.
- [VG88] A. Van Gelder. A satisfiability tester for non-clausal propositional calculus. *Information and Computation*, 79:1–21, 1988.
- [VG07] A. Van Gelder. Another look at graph coloring via propositional satisfiability. *Discrete Applied Mathematics*, 156(2):230–243, 2007.
- [Wal00] T. Walsh. Sat v csp. In *Sixth International Conference on Principles and Practice of Constraint Programming*, pages 441–456. Springer-Verlag, 2000. Lecture Notes in Computer Science vol.

- 1894.
- [War98] J. P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, 68(2):63–69, 1998.
- [WS02] W. Wei and B. Selman. Accelerating random walks. In *Eighth International Conference on Principles and Practice of Constraint Programming*, pages 216–232. Springer, 2002. Lecture Notes in Computer Science vol. 2470.
- [Yok97] M. Yokoo. Why adding more constraints makes a problem easier for hill-climbing algorithms: Analyzing landscapes of csp. In *Third International Conference on Principles and Practice of Constraint Programming*, pages 356–370. Springer-Verlag, 1997. Lecture Notes in Computer Science vol. 1330.

Chapter 3

Complete Algorithms

Adnan Darwiche and Knot Pipatsrisawat

3.1. Introduction

This chapter is concerned with sound and complete algorithms for testing satisfiability, i.e., algorithms that are guaranteed to terminate with a correct decision on the satisfiability/unsatisfiability of the given CNF. One can distinguish between a few approaches on which complete satisfiability algorithms have been based. The first approach is based on existential quantification, where one successively eliminates variables from the CNF without changing the status of its satisfiability. When all variables have been eliminated, the satisfiability test is then reduced into a simple test on a trivial CNF. The second approach appeals to sound and complete inference rules, applying them successively until either a contradiction is found (unsatisfiable CNF) or until the CNF is closed under these rules without finding a contradiction (satisfiable CNF). The third approach is based on systematic search in the space of truth assignments, and is marked by its modest space requirements. The last approach we will discuss is based on combining search and inference, leading to algorithms that currently underly most modern complete SAT solvers.

We start in the next section by establishing some technical preliminaries that will be used throughout the chapter. We will follow by a treatment of algorithms that are based on existential quantification in Section 3.3 and then algorithms based on inference rules in Section 3.4. Algorithms based on search are treated in Section 3.5, while those based on the combination of search and inference are treated in Section 3.6. Note that some of the algorithms presented here could fall into more than one class, depending on the viewpoint used. Hence, the classification presented in Sections 3.3-3.6 is only one of the many possibilities.

3.2. Technical Preliminaries

A *clause* is a disjunction of literals over distinct variables.¹ A propositional sentence is in *conjunctive normal form (CNF)* if it has the form $\alpha_1 \wedge \alpha_2 \wedge \dots \wedge \alpha_n$, where each α_i is a clause. For example, the sentence

¹The insistence that all literals in a clause be over distinct variables is not standard.

$$(A \vee B \vee \neg C) \wedge (\neg A \vee D) \wedge (B \vee C \vee D)$$

is in conjunctive normal form and contains three clauses. Note that according to our definition, a clause cannot contain the literals P and $\neg P$ simultaneously. Hence, a clause can never be valid. Note also that a clause with no literals, the *empty clause*, is inconsistent. Furthermore, a CNF with no clauses is valid.

A convenient way to notate sentences in CNF is using sets. Specifically, a clause $l_1 \vee l_2 \vee \dots \vee l_m$ is expressed as a set of literals $\{l_1, l_2, \dots, l_m\}$. Moreover, a conjunctive normal form $\alpha_1 \wedge \alpha_2 \wedge \dots \wedge \alpha_n$ is expressed as a set of clauses $\{\alpha_1, \alpha_2, \dots, \alpha_n\}$. For example, the CNF given above would be expressed as:

$$\{ \{A, B, \neg C\}, \{\neg A, D\}, \{B, C, D\} \}.$$

This set-based notation will prove very helpful when expressing algorithms that operate on CNFs.

Given our notational conventions, a CNF Δ is valid if Δ is the empty set: $\Delta = \emptyset$. Moreover, a CNF Δ is inconsistent if Δ contains the empty set: $\emptyset \in \Delta$. These two cases correspond to common boundary conditions that arise in recursive algorithms on CNFs.

In general, a formula Δ is said to *imply* another formula Γ , denoted $\Delta \models \Gamma$, iff every assignment that satisfies Δ also satisfies Γ . Note that if a clause C_i is a subset of another clause C_j , $C_i \subseteq C_j$, then $C_i \models C_j$. We say in this case that clause C_i *subsumes* clause C_j , as there is no need to have clause C_j in a CNF that also contains clause C_i . One important fact that we will base our future discussions on is that if a CNF formula implies the empty clause, the formula is essentially unsatisfiable.

3.2.1. Resolution

One of the simplest complete algorithms for testing satisfiability is based on the *resolution* inference rule [Rob65], which is defined as follows. Let P be a Boolean variable, and suppose that Δ is a CNF which contains clauses C_i and C_j , where $P \in C_i$ and $\neg P \in C_j$. The resolution inference rule allows us to derive the clause $(C_i - \{P\}) \cup (C_j - \{\neg P\})$, which is called a *resolvent* that is obtained by *resolving* C_i and C_j . The resolvent of a resolution is a clause implied by the resolved clauses. For example, we can resolve clause $\{A, B, \neg C\}$ with clause $\{\neg B, D\}$ to obtain the resolvent $\{A, \neg C, D\}$. We will say here that $\{A, \neg C, D\}$ is a *B-resolvent* as it results from resolving two clauses on the literals B and $\neg B$. Note that if we can use resolution to derive the empty clause from any formula, it means that the formula implies the empty clause, and, thus, is unsatisfiable.

Resolution is sound but is incomplete in the sense that it is not guaranteed to derive every clause that is implied by the given CNF. However, resolution is *refutation complete* on CNFs, i.e., it is guaranteed to derive the empty clause if the given CNF is unsatisfiable. This result is the basis for using resolution as a complete algorithm for testing satisfiability: we keep applying resolution until either the empty clause is derived (unsatisfiable CNF) or until no more applications of resolution are possible (satisfiable CNF).

Consider now the following resolution trace.

1. $\{\neg P, R\}$
2. $\{\neg Q, R\}$
3. $\{\neg R\}$
4. $\{P, Q\}$

-
5. $\{\neg P\}$ 1, 3
 6. $\{\neg Q\}$ 2, 3
 7. $\{Q\}$ 4, 5
 8. $\{\}$ 6, 7

The clauses before the line represent initial clauses, while clauses below the line represent resolvents, together with the identifiers of clauses used to obtain them. The above resolution trace shows that we can derive the empty clause from the initial set of Clauses (1–4). Hence, the original clauses, together, are unsatisfiable.

An important special case of resolution is called unit resolution. *Unit resolution* is a resolution strategy which requires that at least one of the resolved clauses has only one literal. Such clause is called a *unit clause*. Unit resolution is not refutation complete, which means that it may not derive the empty clause from an unsatisfiable CNF formula. Yet one can apply all possible unit resolution steps in time linear in the size of given CNF. Because of its efficiency, unit resolution is a key technique employed by a number of algorithms that we shall discuss later.

3.2.2. Conditioning

A number of algorithms that we shall define on CNFs make use of the *conditioning* operation. The process of conditioning a CNF Δ on a literal L amounts to replacing every occurrence of literal L by the constant **true**, replacing $\neg L$ by the constant **false**, and simplifying accordingly. The result of conditioning Δ on L will be denoted by $\Delta|L$ and can be described succinctly as follows:

$$\Delta|L = \{\alpha - \{\neg L\} \mid \alpha \in \Delta, L \notin \alpha\}.$$

To further explain this statement, note that the clauses in Δ can be partitioned into three sets:

1. The set of clauses α containing the literal L . Since we are replacing L by **true**, any clause that mentions L becomes satisfied and can be removed from the formula. As a result, these clauses do not appear in $\Delta|L$.
2. The set of clauses α containing the literal $\neg L$. Since we are replacing $\neg L$ by **false**, the occurrences of $\neg L$ in these clauses no longer have any effect. Therefore, these clauses appear in $\Delta|L$ with the literal $\neg L$ removed.
3. The set of clauses α that contain neither L nor $\neg L$. These clauses appear in $\Delta|L$ without change.

For example, if

$$\Delta = \{ \{A, B, \neg C\}, \{\neg A, D\}, \{B, C, D\} \},$$

then

$$\Delta|C = \{ \{A, B\}, \{\neg A, D\} \},$$

and

$$\Delta|\neg C = \{ \{\neg A, D\}, \{B, D\} \}.$$

The definition of conditioning can be extended to multiple literals in the obvious way. For example, $\Delta|CA\neg D = \{\emptyset\}$ (an inconsistent CNF). Moreover, $\Delta|\neg CD = \emptyset$ (a valid CNF).

3.3. Satisfiability by Existential Quantification

We will now discuss a class of complete algorithms which is based on the concept of *existential quantification*. The result of existentially quantifying variable P from a formula Δ is denoted by $\exists P \Delta$ and defined as follows:

$$\exists P \Delta \stackrel{\text{def}}{=} (\Delta|P) \vee (\Delta|\neg P).$$

Consider for example the CNF:

$$\Delta = \{ \{\neg A, B\}, \{\neg B, C\} \},$$

which effectively says that A implies B and B implies C . We then have $\Delta|B = \{\{C\}\}$, $\Delta|\neg B = \{\{\neg A\}\}$ and, hence, $\exists B \Delta = \{\{\neg A, C\}\}$ (i.e., A implies C).

Existential quantification satisfies a number of properties, but the most relevant for our purposes here is this: Δ is satisfiable if and only if $\exists P \Delta$ is satisfiable. Hence, one can replace a satisfiability test on Δ by another satisfiability test on $\exists P \Delta$, which has one fewer variable than Δ (if Δ mentions P). One can therefore existentially quantify all variables in the CNF Δ , one at a time, until we are left with a trivial CNF that contains no variables. This trivial CNF must therefore be either $\{\emptyset\}$, which is unsatisfiable, or $\{\}$, which is valid and, hence, satisfiable.

The above approach for testing satisfiability can be implemented in different ways, depending on how existential quantification is implemented. We will next discuss two such implementations, the first one leading to what is known as the Davis-Putnam algorithm (DP), and the second one leading to what is known as symbolic SAT solving.

3.3.1. The DP Algorithm

The DP algorithm [DP60] for testing satisfiability is based on the following observation. Suppose that Δ is a CNF, and let Γ be another CNF which results from adding to Δ all P -resolvents, and then throwing out all clauses that mention P

(hence, Γ does not mention variable P). It follows in this case that Γ is equivalent to $\exists P\Delta$.² Consider the following CNF,

$$\Delta = \{ \{ \neg A, B \}, \{ \neg B, C \} \}.$$

There is only one B -resolvent in this case: $\{ \neg A, C \}$. Adding this resolvent to Δ , and throwing out those clauses that mention B gives:

$$\Gamma = \{ \{ \neg A, C \} \}.$$

This is equivalent to $\exists B\Delta$ which can be confirmed by computing $\Delta|B \vee \Delta|\neg B$.

The DP algorithm, also known as *directional resolution* [DR94], uses the above observation to existentially quantify all variables from a CNF, one at a time. One way to implement the DP algorithm is using a mechanism known as *bucket elimination* [Dec97], which proceeds in two stages: *constructing and filling* a set of buckets, and then *processing* them in some order. Specifically, given a variable ordering π , we construct and fill buckets as follows:

- A *bucket* is constructed for each variable P and is *labeled* with variable P .
- Buckets are sorted top to bottom by their labels according to order π .
- Each clause α in the CNF is added to the first Bucket P from the top, such that variable P appears in clause α .

Consider for example the CNF

$$\Delta = \{ \{ \neg A, B \}, \{ \neg A, C \}, \{ \neg B, D \}, \{ \neg C, \neg D \}, \{ A, \neg C, E \} \},$$

and the variable order C, B, A, D, E . Constructing and filling buckets leads to:³

$$\begin{aligned} C &: \{ \neg A, C \}, \{ \neg C, \neg D \}, \{ A, \neg C, E \} \\ B &: \{ \neg A, B \}, \{ \neg B, D \} \\ A &: \\ D &: \\ E &: \end{aligned}$$

Buckets are then processed from top to bottom. To process Bucket P , we generate all P -resolvents using only clauses in Bucket P , and then add these resolvents to corresponding buckets below Bucket P . That is, each resolvent α is added to the first Bucket P' below Bucket P , such that variable P' appears in α . Processing Bucket P can then be viewed as a process of existentially quantifying variable P , where the result of such quantifying is now stored in the buckets below Bucket P .

Continuing with the above example, processing Bucket C adds one C -resolvent

²This follows from the fact that prime implicants of the formula can be obtained by closing the formula under resolution. After that, existentially quantifying a variable amounts dropping all clauses that mention the variable. Interested readers are referred to [LLM03, Mar00].

³It is not uncommon for buckets to be empty. It is also possible for all clauses to fall in the same bucket.

Algorithm 3.1 DP(CNF Δ , variable order π): returns UNSATISFIABLE or SATISFIABLE.

```

1: for each variable  $V$  of  $\Delta$  do
2:   create empty bucket  $B_V$ 
3: for each clause  $C$  of  $\Delta$  do
4:    $V =$  first variable of  $C$  according to order  $\pi$ 
5:    $B_V = B_V \cup \{C\}$ 
6: for each variable  $V$  of  $\Delta$  in order  $\pi$  do
7:   if  $B_V$  is not empty then
8:     for each  $V$ -resolvent  $C$  of clauses in  $B_V$  do
9:       if  $C$  is the empty clause then
10:        return UNSATISFIABLE
11:        $U =$  first variable of clause  $C$  according to order  $\pi$ 
12:        $B_U = B_U \cup \{C\}$ 
13: return SATISFIABLE

```

to Bucket A :

$$\begin{aligned}
 C &: \{\neg A, C\}, \{\neg C, \neg D\}, \{A, \neg C, E\} \\
 B &: \{\neg A, B\}, \{\neg B, D\} \\
 A &: \{\neg A, \neg D\} \\
 D &: \\
 E &:
 \end{aligned}$$

The buckets below Bucket C will now contain the result of existentially quantifying variable C . Processing Bucket B adds one B -resolvent to Bucket A :

$$\begin{aligned}
 C &: \{\neg A, C\}, \{\neg C, \neg D\}, \{A, \neg C, E\} \\
 B &: \{\neg A, B\}, \{\neg B, D\} \\
 A &: \{\neg A, \neg D\}, \{\neg A, D\} \\
 D &: \\
 E &:
 \end{aligned}$$

At this stage, the buckets below Bucket B contain the resulting of existentially quantifying both variables C and B . Processing Bucket A , leads to no new resolvents. We therefore have $\exists C, B, A \Delta = \{\}$ and the original CNF is consistent.

Algorithm 3.1 contains the pseudocode for directional resolution. Note that the amount of work performed by directional resolution is quite dependent on the chosen variable order. For example, considering the same CNF used above with the variable order E, A, B, C, D leads to the following buckets:

$$\begin{aligned}
 E &: \{A, \neg C, E\} \\
 A &: \{\neg A, B\} \{\neg A, C\} \\
 B &: \{\neg B, D\} \\
 C &: \{\neg C, \neg D\} \\
 D &:
 \end{aligned}$$

Processing the above buckets yields no resolvents in this case! This is another proof for the satisfiability of the given CNF, which was obtained by doing less work due to the chosen variable order.

In the case that the formula is satisfiable, we can extract a satisfying assignment from the trace of directional resolution in time that is linear to the size of

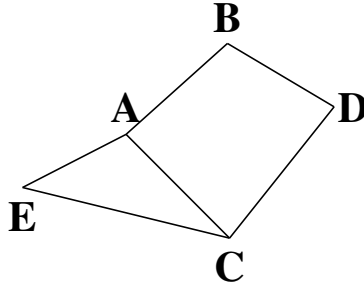


Figure 3.1. A connectivity graph for a CNF.

the formulas in all buckets [DR94]. To do so, we process the variables in the reverse order of the elimination (bottom bucket to top bucket). Let the elimination order be $\pi = V_1, V_2, \dots, V_n$. For each variable V_i , if its bucket is empty, it can be assigned any value. Otherwise, we assign to V_i the value that, together with the values of $V_j, i < j < n$, satisfies all clauses in its bucket.

Using a suitable variable order, the time and space complexity of directional resolution can be shown to be $O(n \exp(w))$ [DR94], where n is the size of CNF Δ and w is the treewidth of its *connectivity graph*: an undirected graph G over the variables of Δ , where an edge exists between two variables in G iff these variables appear in the same clause in Δ . Figure 3.1 depicts the connectivity graph for CNF we considered earlier:

$$\Delta = \{ \{\neg A, B\}, \{\neg A, C\}, \{\neg B, D\}, \{\neg C, \neg D\}, \{A, \neg C, E\} \}.$$

The complexity of this algorithms is discussed in more details in [DR94], which also shows that directional resolution becomes tractable on certain classes of CNF formula and that it can be used to answer some entailment queries on the formula.

3.3.2. Symbolic SAT Solving

The main problem observed with the DP algorithm is its space complexity, as it tends to generate too many clauses. A technique for dealing with this space complexity is to adopt a more compact representation of the resulting clauses. One such representation is known as *Binary Decision Diagrams (BDDs)* whose adoption leads to the class of *symbolic SAT algorithms* [CS00, AV01, MM02, PV04, FKS⁺04, JS04, HD04, SB06]. Many variations of this approach have been proposed over the years, but we discuss here only the basic underlying algorithm of symbolic SAT solving.

Figure 3.2 depicts an example BDD over three variables together with the models it encodes. Formally, a BDD is a rooted directed acyclic graph where there are at most two sinks, labeled with 0 and 1 respectively, and every other node is labeled with a Boolean variable and has exactly two children, distinguished as *low* and *high* [Bry86]. A BDD represents a propositional sentence whose models can be enumerated by taking all paths from the root to the 1-sink: taking the low

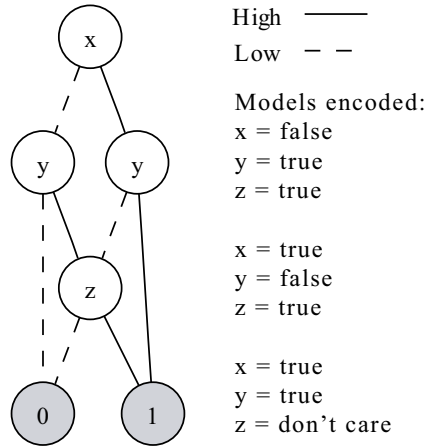


Figure 3.2. A BDD and the models it encodes.

(high) child of a node labeled with a variable X corresponds to assigning **false** (**true**) to X ; in case X does not appear on the path, it is marked as “don’t care” which means either value can be assigned.

In practice, BDDs are usually maintained so that variables appear in the same order on any path from the root to a sink, leading to Ordered BDDs (OBDDs). Two additional properties are often imposed: that there is no node whose two children are identical, and that there are no isomorphic sub-graphs. Under these conditions OBDDs are known to be a canonical form, meaning that there is a unique OBDD for any propositional sentence under a given variable order [Bry86]. Moreover, any binary operation on two OBDDs with the same variable order can be carried out using the *Apply* algorithm [Bry86], whose complexity is linear in the product of the operand sizes. Hence, one can existentially quantify a variable X from an OBDD by first conditioning it on X and then on $\neg X$, each of which can be done in time linear in the OBDD size. One can then disjoin the resulting OBDDs to obtain the result of existential quantification. The whole process can therefore be implemented in time which is quadratic in the size of the OBDD.

To solve SAT, one may convert the CNF formula into the equivalent OBDD and existentially quantify all variables from it. However, this naïve approach is hardly practical as the size of the OBDD will be too large. Therefore, to make this approach viable, symbolic SAT solving appeals to the following property of existential quantification:

$$\exists X \Delta = (\exists X \Gamma_X) \wedge \Gamma,$$

where $\Delta = \Gamma_X \wedge \Gamma$ and Γ_X contains all clauses of CNF Δ that mention variable X . Symbolic SAT solving starts by converting each clause C into a corresponding OBDD, $OBDD(C)$, a process which can be done in time linear in the clause size. To existentially quantify variable X , all OBDDs $\Gamma_1, \dots, \Gamma_n$ that mention variable X are conjoined together to produce one OBDD Γ_X , from which X is existentially quantified. This technique is called *early quantification* [BCM⁺90,

Algorithm 3.2 SYMBOLIC_SAT(CNF Δ , variable order π): returns UNSATISFIABLE or SATISFIABLE.

```

1: for each variable  $V$  of  $\Delta$  do
2:   create empty bucket  $B_V$ 
3: for each clause  $C$  of  $\Delta$  do
4:    $V =$  first variable of  $C$  according to order  $\pi$ 
5:    $B_V = B_V \cup \{OBDD(C)\}$ 
6: for each variable  $V$  of  $\Delta$  according to order  $\pi$  do
7:   if  $B_V$  is not empty then
8:      $\Gamma_V =$  conjunction of all OBDDs in  $B_V$ 
9:     if  $\Gamma_V =$  zero then
10:      return UNSATISFIABLE
11:      $\Gamma_V = \exists V \Gamma_V$ 
12:      $U =$  first variable of  $\Gamma_V$  according to order  $\pi$ 
13:      $B_U = B_U \cup \{\Gamma_V\}$ 
14: return SATISFIABLE

```

HKB96, TSL⁺90]. The resulting OBDD $\exists X \Gamma_X$ will then replace the OBDDs $\Gamma_1, \dots, \Gamma_n$, and the process is continued. If a zero-OBDD (one which is equivalent to false) is produced in the process, the algorithm terminates while declaring the original CNF unsatisfiable. However, if all variables are eliminated without producing a zero-OBDD, the original CNF is declared satisfiable.

The strategy for scheduling conjunctions and variable quantifications plays an important role in the performance of this algorithm. One goal is to try to minimize the size of intermediate OBDDs to reduce space requirement, which is often a major problem. Unfortunately, this problem is known to be NP-Hard [HKB96]. In [HD04], bucket elimination has been studied as a scheduling strategy. Algorithm 3.2 illustrates a symbolic SAT algorithm with bucket elimination. In [HD04], recursive decomposition [Dar01] and min-cut linear arrangement [AMS01] have been used as methods for ordering variables. Another approach for scheduling quantification is clustering [RAB⁺95]. In [PV04], this approach was studied together with the maximum cardinality search (MCS) [TY84] as a variable ordering heuristic.

3.4. Satisfiability by Inference Rules

We have already discussed in Section 3.2 the use of the resolution inference rule in testing satisfiability. We will now discuss two additional approaches based on the use of inference rules.

3.4.1. Stålmarck's Algorithm

Stålmarck's algorithm was originally introduced in 1990 [SS90], and later patented in 1994 [Stå94], as an algorithm for checking tautology of arbitrary propositional formulas (not necessarily in CNF). To test the satisfiability of a CNF, we can equivalently check whether its negation is a tautology using Stålmarck's algorithm [SS90, Wid96, Har96]. Here, we will discuss Stålmarck's algorithm in its original form—as a tautology prover.

The algorithm starts with a preprocessing step in which it transforms implications into disjunctions, removes any double negations and simplifies the formula by applying rudimentary inference rules (such as removing **true** from conjunctions and **false** from disjunctions). If preprocessing is able to derive the truth value of the formula (as **true** or **false**), then the algorithm terminates. Otherwise, the algorithm continues on to the next step.

The preprocessed formula is converted into a conjunction of “triplets.” Each triplet has the form $p \Leftrightarrow (q \otimes r)$, where \otimes is either a conjunction or an equivalence and p must be a Boolean variable, while r and q are literals. During this process, new Boolean variables may need to be introduced to represent sub-formulas. For example, the formula $\neg((a \Leftrightarrow b \wedge c) \wedge (b \Leftrightarrow \neg c) \wedge a)$ may be transformed into

$$\begin{aligned} v_1 &\Leftrightarrow (b \wedge c) \\ v_2 &\Leftrightarrow (a \Leftrightarrow v_1) \\ v_3 &\Leftrightarrow (b \Leftrightarrow \neg c) \\ v_4 &\Leftrightarrow (v_3 \wedge a) \\ v_5 &\Leftrightarrow (v_2 \wedge v_4) \end{aligned}$$

Here, $\neg v_5$ is a literal whose truth value is equivalent to that of the original formula.⁴ This conversion is meant to allow the algorithm to apply inference rules efficiently and to partially decompose the original formula into smaller sub-formulas that involve fewer variables. After this transformation, the algorithm assumes, for the sake of contradiction, that the whole formula has truth value **false**. For the remaining part, the algorithm tries to derive a contradiction, which would show that the original formula was actually a tautology.

In order to achieve this, the algorithm applies a set of inference rules called *simple rules* (or *propagation rule*) to the triplets to obtain more conclusions. Each conclusion either assigns a value to a variable or ties the value of 2 literals together. Some examples of simple rules are

$$\begin{aligned} p \Leftrightarrow (q \wedge r) &:\text{if } p = \neg q \text{ then } q = \mathbf{true} \text{ and } r = \mathbf{false} \\ p \Leftrightarrow (q \wedge r) &:\text{if } p = \mathbf{true} \text{ then } q = \mathbf{true} \text{ and } r = \mathbf{true} \\ p \Leftrightarrow (q \Leftrightarrow r) &:\text{if } p = q \text{ then } r = \mathbf{true} \\ p \Leftrightarrow (q \Leftrightarrow r) &:\text{if } p = \mathbf{true} \text{ then } q = r. \end{aligned}$$

The reader is referred to [Har96] for a complete set of simple rules used by the algorithm. The process of repeatedly and exhaustively applying simple rules to the formula is called *0-saturation*. After 0-saturation, if the truth value of the formula can be determined, then the algorithm terminates. Otherwise, the algorithm proceeds by applying the *dilemma rule*.

⁴This example is taken from [Har96].

The dilemma rule is a way of deducing more conclusions about a formula by making assumptions, which aid the application of simple rules. The rule operates as follows. Let Δ be the formula after 0-saturation. For each Boolean variable v in Δ , the algorithm 0-saturates $\Delta|v$ and $\Delta|\neg v$ to produce conclusions Γ_v and $\Gamma_{\neg v}$, respectively. Then, all conclusions that are common between Γ_v and $\Gamma_{\neg v}$ are added to Δ . In other words, the algorithm tries both values of the variable v and keeps all the conclusions derivable from both branches. These conclusions necessarily hold regardless of the value of v . The process is repeated for all variables until no new conclusions can be derived. The algorithm terminates as soon as contradicting conclusions are added to the knowledge base or the truth value of the formula can be determined from the conclusions. The process of applying the dilemma rule exhaustively on all variables is called 1-saturation. As one might expect, the algorithm can apply the dilemma rule with increasing depths of assignment. In particular, n -saturation involves case-splitting over all combinations of n variables simultaneously and gathering common conclusions. If we allow n to be large enough, the algorithm is guaranteed to either find a contradiction or reach a satisfying assignment. Because Stålmarck's algorithm n -saturates formulas starting from $n = 0$, it is oriented toward finding short proofs for the formula's satisfiability or unsatisfiability.

3.4.2. HeerHugo

HeerHugo [GW00] is a satisfiability prover that was largely inspired by Stålmarck's algorithm. Although, strictly speaking, HeerHugo is not a tautology prover, many techniques used in this algorithm have lots of similarity to those of Stålmarck's algorithm.

According to this algorithm, the input formula is first converted into CNF with at most three literals per clause. Then, HeerHugo applies a set of basic inference rules, which are also called simple rules, to the resulting formula. Although these inference rules share the name with those rules in Stålmarck's algorithm, the two set of rules are rather different. Simple rules used in HeerHugo appear to be more powerful. They include unit resolution, subsumption, and a restricted form of resolution (see [GW00] for more details).

Resolution is selectively performed by HeerHugo, in order to existentially quantify some variables. This technique is similar to the one used in DP [DP60]. However, in HeerHugo, resolution is only carried out when it results in a smaller formula. Because of these rules, clauses are constantly added and removed from the knowledge base. Consequently, HeerHugo incorporates some subsumption and ad-hoc resolution rules for removing subsumed clauses and opportunistically generating stronger clauses.

During the process of applying simple rules, conclusions may be drawn from the formula. Like in Stålmarck's algorithm, a conclusion is either an assignment of a variable or an equivalence constraint between two literals. If a contradiction is found during this step, the formula is declared unsatisfiable. Otherwise, the algorithm continues with the *branch/merge* rule, which is essentially the same as the dilemma rule of Stålmarck's. The algorithm also terminates as soon as the truth value of the formula can be determined from the conclusions. The

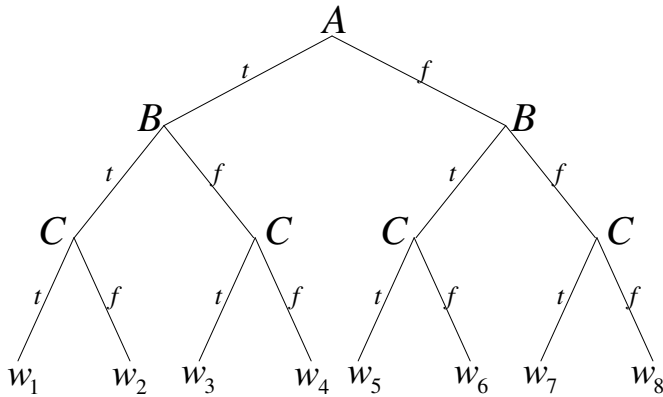


Figure 3.3. A search tree for enumerating all truth assignments over variables A , B and C .

completeness of HeerHugo is achieved by successively increasing the number of variables in the branch/merge rule, in a manner that is similar to applying n -saturation in Stålmarck's.

3.5. Satisfiability by Search: The DPLL Algorithm

We will discuss in this section the DPLL algorithm [DLL62], which is based on systematic search in the space of truth assignments. This algorithm is marked by its modest space requirements and was developed in response to the challenges posed by the space requirements of the DP algorithm.

Figure 3.3 depicts a search tree that enumerates the truth assignments over three variables. The first observation about this tree is that its leaves are in one-to-one correspondence with the truth assignments under consideration. Hence, testing satisfiability can be viewed as a process of searching for a leaf node that satisfies the given CNF. The second observation is that the tree has a finite depth of n , where n is the number of Boolean variables. Therefore, it is best to explore the tree using depth-first-search as given in Algorithm 3.3, which is called initially with depth $d = 0$. The algorithm makes use of the conditioning operator on CNFs, which leads to simplifying the CNF by either removing clauses or reducing their size.

Consider now the CNF:

$$\Delta = \{ \{-A, B\}, \{-B, C\} \},$$

and the search node labeled with **F** in Figure 3.4. At this node, Algorithm 3.3 will condition Δ on literals $A, \neg B$, leading to:

$$\Delta|A, \neg B = \{ \{\text{false}, \text{false}\}, \{\text{true}, C\} \} = \{ \{ \} \}.$$

Algorithm 3.3 DPLL-(CNF Δ , depth d): returns a set of literals or UNSATISFIABLE. Variables are named P_1, P_2, \dots

```

if  $\Delta = \{\}$  then
  return  $\{\}$ 
else if  $\{\} \in \Delta$  then
  return UNSATISFIABLE
else if  $\mathbf{L} = \text{DPLL}(\Delta | P_{d+1}, d+1) \neq \text{UNSATISFIABLE}$  then
  return  $\mathbf{L} \cup \{P_{d+1}\}$ 
else if  $\mathbf{L} = \text{DPLL}(\Delta | \neg P_{d+1}, d+1) \neq \text{UNSATISFIABLE}$  then
  return  $\mathbf{L} \cup \{\neg P_{d+1}\}$ 
else
  return UNSATISFIABLE

```

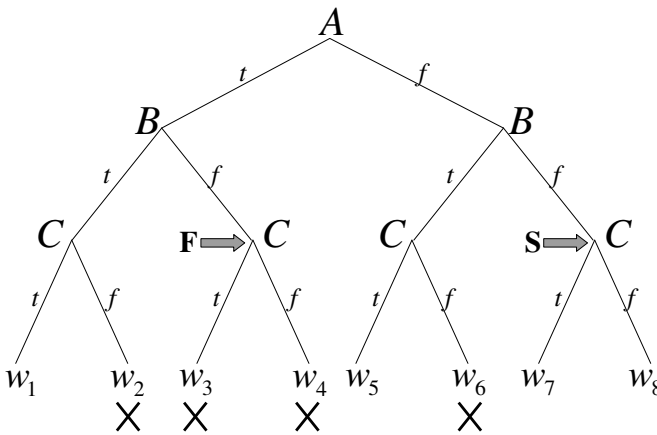


Figure 3.4. A search tree for that enumerates all truth assignments over variables A , B and C . Assignments marked by \times are not models of the CNF $\{\neg A, B\}, \{\neg B, C\}$.

Hence, the algorithm will immediately conclude that neither truth assignment ω_3 nor ω_4 are models of Δ . The ability to detect contradictions at internal nodes in the search is quite significant as it allows one to dismiss all truth assignments that are characterized by that node, without having to visit each one of them explicitly. Algorithm 3.3 can also detect success at an internal node, which implies that all truth assignments characterized by that node are models of the CNF. Consider for example the internal node labelled with **S** in Figure 3.4. This node represents truth assignments ω_7 and ω_8 , which are both models of Δ . This can be detected by conditioning Δ on literals $\neg A, \neg B$:

$$\Delta | \neg A, \neg B = \{ \{\text{true}, \text{false}\}, \{\text{true}, C\} \} = \{\}.$$

Hence, all clauses are subsumed immediately after we set the values of A and B to false, and regardless of how we set the value of C . This allows us to conclude that both ω_7 and ω_8 are models of Δ , without having to inspect each of them

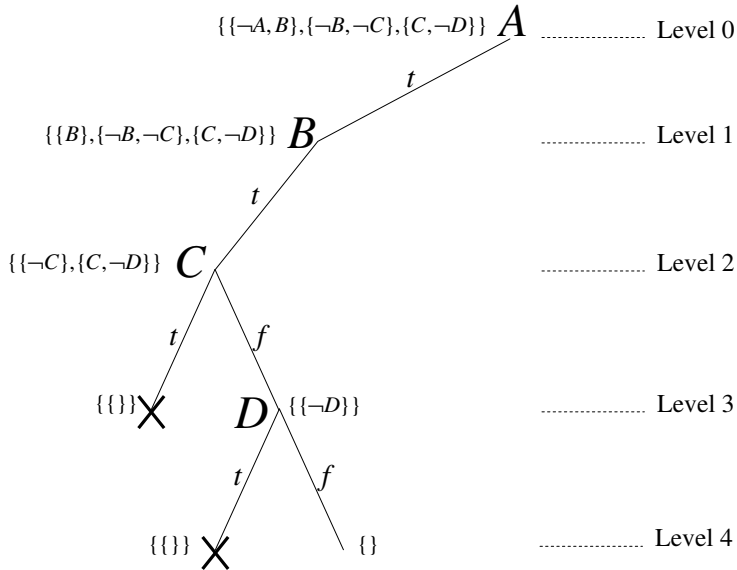


Figure 3.5. A termination tree, where each node is labelled by the corresponding CNF. The last node visited during the search is labelled with $\{\}$. The label \times indicates the detection of a contradiction at the corresponding node.

individually. Given that Algorithm 3.3 may detect success before reaching a leaf node, it may return less than n literals, which would characterize a set of truth assignments in this case, all of which are guaranteed to be models of the input formula.

3.5.1. Termination Trees

One way to measure the amount of work done by Algorithm 3.3 is using the notion of a *termination tree*: the subset of search tree that is actually explored during search. Figure 3.5 depicts an example termination tree for the CNF:

$$\Delta = \{ \{ \neg A, B \}, \{ \neg B, \neg C \}, \{ C, \neg D \} \}.$$

According to this figure, Δ is satisfiable and the truth assignment $A, B, \neg C, \neg D$ is a satisfying assignment. The figure also provides a trace of Algorithm DPLL (Algorithm 3.3) as it shows the conditioned CNF at each node visited during the search process. The nodes in a termination tree belong to different levels as shown in Figure 3.5, with the root node being at Level 0. Termination trees are also useful in characterizing different satisfiability problems since the size and depth of tree appear to be good indicators of the problem character and difficulty.

3.5.2. Unit Resolution

Consider the termination tree of Figure 3.5, and corresponding CNF:

$$\Delta = \{ \{ \neg A, B \}, \{ \neg B, \neg C \}, \{ C, \neg D \} \}.$$

Consider also the node at Level 1, which results from setting variable A to true, and its corresponding CNF:

$$\Delta|A = \{ \{ B \}, \{ \neg B, \neg C \}, \{ C, \neg D \} \}.$$

This CNF is neither empty, nor contains the empty clause. Therefore, we cannot yet declare early success or early failure, which is why Algorithm 3.3 continues searching below Level 1 as shown in Figure 3.5. We will show now, however, that by employing unit resolution, we can indeed declare success and end the search at this node.

The *unit resolution technique* (or *unit propagation*) is quite simple: Before we perform the tests for success or failure on Lines 1 and 2 of Algorithm 3.3, we first close the CNF under unit resolution and collect all unit clauses in the CNF. We then assume that variables are set to satisfy these unit clauses. That is, if the unit clause $\{P\}$ appears in the CNF, we set P to true. And if the unit clause $\{\neg P\}$ appears in the CNF, we set P to false. We then simplify the CNF given these settings and check for either success (all clauses are subsumed) or failure (the empty clause is derived).

To incorporate unit resolution into our satisfiability algorithms, we will introduce a function UNIT-RESOLUTION, which applies to a CNF Δ and returns two results:

- \mathbf{I} : a set of literals that were either present as unit clauses in Δ , or were derived from Δ by unit resolution.
- Γ : a new CNF which results from conditioning Δ on literals \mathbf{I} .

For example, if the CNF

$$\Delta = \{ \{ \neg A, \neg B \}, \{ B, C \}, \{ \neg C, D \}, \{ A \} \},$$

then $\mathbf{I} = \{A, \neg B, C, D\}$ and $\Gamma = \{\}$. Moreover, if

$$\Delta = \{ \{ \neg A, \neg B \}, \{ B, C \}, \{ \neg C, D \}, \{ C \} \},$$

then $\mathbf{I} = \{C, D\}$ and $\Gamma = \{ \{ \neg A, \neg B \} \}$. Unit resolution is a very important component of search-based SAT solving algorithms. Part 1, Chapter 4 discusses in details the modern implementation of unit resolution employed by many SAT solvers of this type.

Algorithm DPLL, for Davis, Putnam, Logemann and Loveland, is a refinement on Algorithm 3.3 which incorporates unit resolution. Beyond applying unit resolution on Line 1, we have two additional changes to Algorithm 3.3. First of all, we no longer assume that variables are examined in the same order as we go down the search tree. Secondly, we no longer assume that a variable is set to true first, and then to false; see Line 7. The particular choice of a literal L on Line 7

Algorithm 3.4 DPLL(CNF Δ): returns a set of literals or UNSATISFIABLE.

```

1:  $(\mathbf{I}, \Gamma) = \text{UNIT-RESOLUTION}(\Delta)$ 
2: if  $\Gamma = \{\}$  then
3:   return  $\mathbf{I}$ 
4: else if  $\{\} \in \Gamma$  then
5:   return UNSATISFIABLE
6: else
7:   choose a literal  $L$  in  $\Gamma$ 
8:   if  $\mathbf{L} = \text{DPLL}(\Gamma|L) \neq \text{UNSATISFIABLE}$  then
9:     return  $\mathbf{L} \cup \mathbf{I} \cup \{L\}$ 
10:  else if  $\mathbf{L} = \text{DPLL}(\Gamma|\neg L) \neq \text{UNSATISFIABLE}$  then
11:    return  $\mathbf{L} \cup \mathbf{I} \cup \{\neg L\}$ 
12:  else
13:    return UNSATISFIABLE

```

can have a dramatic impact on the running time of DPLL. Part 1, Chapter 8 is dedicated to heuristics for making this choice, which are known as *variable ordering* or *splitting* heuristics when choosing a variable, and *phase selection* heuristics when choosing a particular literal of that variable.

3.6. Satisfiability by Combining Search and Inference

We will now discuss a class of algorithms for satisfiability testing which form the basis of most modern complete SAT solvers. These algorithms are based on the DPLL algorithm, and have undergone many refinements over the last decade, making them the most efficient algorithms discussed thus far. Yet, these refinements have been significant enough to change the behavior of DPLL to the point where the new algorithms are best understood in terms of an interplay between search and inference. Early successful solvers employing this approach, such as GRASP [MSS99] and Rel_{sat} [BS97], gave rise to modern solvers namely BerkMin [GN02], JeruSAT [DHN05a], MiniSAT [ES03], PicoSAT [Bie07], Rsat [PD07], Siege [Rya04], TiniSAT [Hua07a], and zChaff [MMZ⁺01]. We start this next section by discussing a main limitation of the DPLL algorithm, which serves as a key motivation for these refinements.

3.6.1. Chronological Backtracking

Algorithm DPLL is based on *chronological backtracking*. That is, if we try both values of a variable at level l , and each leads to a contradiction, we move to level $l - 1$, undoing all intermediate assignments in the process, and try another value at that level (if one remains to be tried). If there are no other values to try at level $l - 1$, we move to level $l - 2$, and so on. If we move all the way to Level 0, and each value there leads to a contradiction, we know that the CNF is inconsistent.

The process of moving from the current level to a lower level is known as *backtracking*. Moreover, if backtracking to level l is done only after having tried both values at level $l + 1$, then it is called *chronological backtracking*, which is the kind of backtracking used by DPLL. The problem with this type of backtracking is that it does not take into account the information of the contradiction that triggers the backtrack.

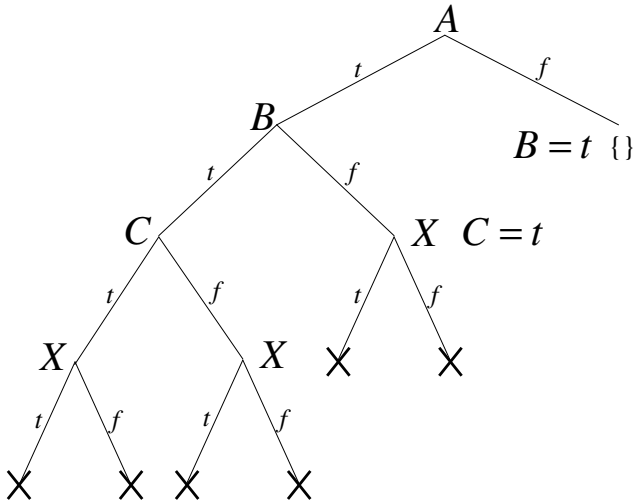


Figure 3.6. A termination tree. Assignments shown next to nodes are derived using unit resolution.

To consider a concrete example, let us look at how standard DPLL behaves on the following CNF, assuming a variable ordering of A, B, C, X, Y, Z :

$$\Delta = \begin{array}{l} 1. \{A, B\} \\ 2. \{B, C\} \\ 3. \{\neg A, \neg X, Y\} \\ 4. \{\neg A, X, Z\} \\ 5. \{\neg A, \neg Y, Z\} \\ 6. \{\neg A, X, \neg Z\} \\ 7. \{\neg A, \neg Y, \neg Z\} \end{array} \quad (3.1)$$

Figure 3.6 depicts the termination tree for DPLL on the above CNF. We start first by setting A to true and explore the subtree below that node, only to find that all branches lead to contradictions. This basically means that $A = \text{true}$ is not a part of any solution. In other words, the formula $\Delta \wedge A$ is inconsistent. Note, however, that DPLL is unable to discover this fact in the knowledge until it has set variables B, C and X . Recall that DPLL uses unit resolution, which is not refutation complete. This explains why DPLL cannot detect the contradiction early on.

To provide more examples of the previous phenomena, note that DPLL is able to detect a contradiction in $\Delta|A, B, C, X$ and in $\Delta|A, B, C, \neg X$, which immediately implies a contradiction in $\Delta|A, B, C$. Yet DPLL is unable to detect a contradiction in $\Delta|A, B, C$.

When DPLL detects the contradiction in $\Delta|A, B, C, X$, it backtracks and tries the other value of X , leading also to a contradiction in $\Delta|A, B, C, \neg X$. DPLL then backtracks again and tries the second value of variable C . Again, no contradiction

is detected by DPLL in $\Delta|A, B, \neg C$, yet it later discovers that both $\Delta|A, B, \neg C, X$ and $\Delta|A, B, \neg C, \neg X$ are contradictory, leading it to backtrack again, and so on.

The key point here is that all these contradictions that are discovered deep in the search tree are actually caused by having set A to `true` at Level 0. That is, the settings of variables B and C at Levels 1 and 2 are irrelevant here. However, chronological backtracking is not aware of this fact. As a result, it tries different values of variables B, C , hoping to fix the contradiction. As shown in Figure 3.6, DPLL with chronological backtracking encounters 6 contradictions before realizing that $A = \text{true}$ was a wrong decision. In general, there can be many irrelevant variables between the level of conflict and the real cause of the conflict. Chronological backtracking may lead the solver to repeat the same conflict over and over again in different settings of irrelevant variables.

3.6.2. Non-Chronological Backtracking

Non-chronological backtracking addresses this problem by taking into account the set of variables that actually are involved in the contradiction. Originally proposed as a technique for solving constraint satisfaction problems (CSPs), *non-chronological backtracking* is a method for the solver to quickly get out of the portion of the search space that contains no solution [SS77, Gas77, Pro93, BS97]. This process involves backtracking to a lower level l without necessarily trying every possibility between the current level and l .⁵ Non-chronological backtracking is sometimes called for as it is not uncommon for the contradiction discovered at level l to be caused by variable settings that have been committed at levels much lower than l . In such a case, trying another value at level l , or at level $l - 1$, may be fruitless.

Non-chronological backtracking can be performed by first identifying every assignment that contributes to the derivation of the empty clause [BS97]. This set of assignments is referred to as the *conflict set* [Dec90]. Then, instead of backtracking to the last unflipped decision variable as in the case of chronological backtracking, non-chronological backtracking backtracks to the most recent decision variable that appears in the conflict set and tries its different value. Note that, during this process, all intermediate assignments (between the current level and the backtrack level) are erased.

In the above example, after the algorithm assigns $A = \text{true}, B = \text{true}, C = \text{true}, X = \text{true}$, unit resolution will derive $Y = \text{true}$ (Clause 3), $Z = \text{true}$ (Clause 5), and detect that Clause 7 becomes empty (all literals are already false). In this case, the conflict set is $\{A = \text{true}, X = \text{true}, Y = \text{true}, Z = \text{true}\}$. Note that B and C do not participate in this conflict. Non-chronological backtracking will backtrack to try a different value of X —the most recent decision variable in the conflict set. After setting $X = \text{false}$, the algorithm will detect another conflict. This time, the conflict set will be $\{A = \text{true}, X = \text{true}, Z = \text{true}\}$. Since we have exhausted the values for X , non-chronological backtracking will now backtrack to A and set $A = \text{false}$ and continue. Note that, this time, the algorithm is able

⁵There are several variations of non-chronological backtracking used in CSP and SAT [Gas77, Pro93, MSS99, BS97, ZMMM01].

to get out of the conflict without trying different values of B or C . Moreover, it only ran into 2 contradictions in the process.

Non-chronological backtracking partially prevents the algorithm from repeating the same conflict. However, as soon as the algorithm backtracks past every variable in the conflict set (possibly due to a different conflict), it can still repeat the same mistake in the future.

One can address this problem by empowering unit resolution through the addition of clauses to the CNF. For example, the clause $\{\neg A, \neg X\}$ is implied by the above CNF. Moreover, if this clause were present in the CNF initially, then unit resolution would have discovered a contradiction immediately after setting A to true. This leaves the question of how to identify such clauses in the first place. As it turns out, each time unit resolution discovers a contradiction, there is an opportunity to identify a clause which is implied by the CNF and which would be allow unit resolution to realize new implications. This clause is known as a *conflict-driven clause* (or *conflict clause*) [MSS99, BS97, ZMMM01, BKS04]. Adding it to the CNF will allow unit resolution to detect this contradiction earlier and to avoid the same mistake in the future.⁶

3.6.3. Non-Chronological Backtracking and Conflict-Driven Clauses

The use of non-chronological backtracking in modern SAT solvers is tightly coupled with the derivation of conflict-driven clauses. We will now discuss the method that modern SAT solvers use to derive conflict-driven clauses from conflicts and the specific way they perform non-chronological backtracking. The combination of these techniques makes sure that unit resolution is empowered every time a conflict arises and that the solver will not repeat any mistake. The identification of conflict-driven clauses is done through a process known as *conflict analysis*, which analyzes a trace of unit resolution known as the *implication graph*.

3.6.3.1. Implication Graph

An implication graph is a bookkeeping device that allows us to record dependencies among variable settings as they are established by unit resolution. Figure 3.7 depicts two implication graphs for the knowledge base in (3.1). Figure 3.7(a) is the implication graph after setting variables A, B, C and X to true.

Each node in an implication graph has the form $l/V=v$, which means that variable V has been set to value v at level l . Note that a variable is set either by a *decision* or by an *implication*. A variable is set by an implication if the setting is due to the application of unit resolution. Otherwise, it is set by a decision.

Whenever unit resolution is used to derive a variable assignment, it uses a clause and a number of other variable assignments to justify the implication. In this case, we must have an edge into the implied assignment from each of the assignments used to justify the implication. For example, in the implication graph of Figure 3.7(a), variable Y is set to true at Level 3 by using Clause 3 and the variable settings $A=\text{true}$ and $X=\text{true}$. Hence, we have an edge from $A=\text{true}$

⁶The idea of learning from conflicts originated from the successful applications in CSP [SS77, Gen84, Dav84, Dec86, dKW87].

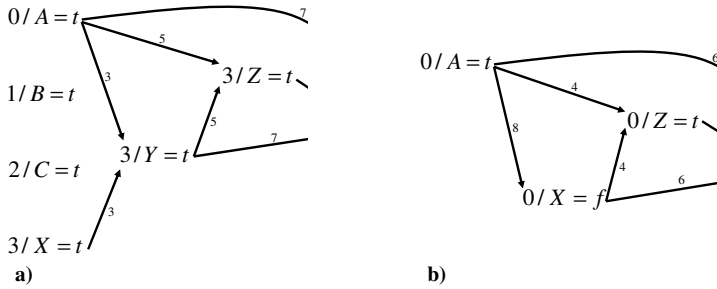


Figure 3.7. Two implication graphs.

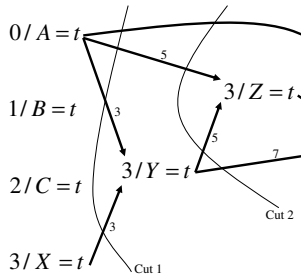


Figure 3.8. Three cuts in an implication graph, leading to three conflict sets.

to $Y=\text{true}$, and another from $X=\text{true}$ to $Y=\text{true}$. Moreover, both of these edges are labeled with 3, which is the ID of clause used in implying $Y=\text{true}$.

An implication graph may also have a special node representing a contradiction derived by unit resolution. This is indicated by $\{\}$ in the implication graph, with its parents being all variable settings that were used in deriving an empty clause. For example, in Figure 3.7(a), the settings $A=\text{true}$, $Y=\text{true}$ and $Z=\text{true}$ were used with Clause 7 to derive the empty clause.

3.6.3.2. Deriving a Conflict-Driven Clause

The process of deriving a conflict-driven clause involves identifying a conflict set from the implication graph and converting this set into a clause. A conflict set can be obtained in the following manner. Every *cut* in the implication graph defines a conflict set as long as that cut separates the decision variables (root nodes) from the contradiction (a leaf node) [MSS99, ZMMM01, Rya04].⁷ Any node (variable assignment) with an outgoing edge that crosses the cut will then be in the conflict set. Intuitively, a conflict set contains assignments that are sufficient to cause the conflict. Figure 3.8 depicts a few cuts in the implication graph of Figure 3.7(a), leading to conflict sets $\{A=\text{true}, X=\text{true}\}$, $\{A=\text{true}, Y=\text{true}\}$ and $\{A=\text{true}, Y=\text{true}, Z=\text{true}\}$.

⁷Note that we may have multiple, disconnected components of the implication graph at any point in the search. Our analysis always concerns the component which contains the contradiction node.

Given the multiplicity of these conflict sets, one is interested in choosing the most useful one. In practice, virtually all modern SAT solvers insist on selecting cuts that involve exactly one variable assigned at the level of conflict for a reason that we become apparent later. For the implication graph in Figure 3.7(a), the sets $\{Y = \text{true}, A = \text{true}\}$ and $\{X = \text{true}, A = \text{true}\}$ meet this criterion. Similarly, for the graph in Figure 3.7(b), $\{A = \text{true}\}$ is a conflict cut with such property. An actual implementation that analyzes the implication graph for the right conflict set with desired properties will be described in Part 1, Chapter 4.

Once a conflict set is identified, a conflict-driven clause can be obtained by simply negating the assignments in the set. For example, if the conflict set is $\{A = \text{true}, B = \text{false}, C = \text{false}\}$, then the conflict-driven clause derived is $\{\neg A, B, C\}$. This clause represents the previously-implicit constraint that the assignments in the conflict set cannot be made simultaneously ($A = \text{true}, B = \text{false}, C = \text{true}$ will now violate this new clause). Conflict-driven clauses generated from cuts that contain exactly one variable assigned at the level of conflict are said to be *asserting* [ZMMM01]. Modern SAT solvers insist on learning only asserting clauses. We will discuss later the properties of asserting clauses that make them useful and even necessary for the completeness of the algorithm. From now on, we assume that every conflict-driven clause is asserting, unless stated otherwise.

3.6.3.3. Learning a Conflict-Driven Clause and Backtracking

Once the conflict-driven clause is derived from the implication graph, it is added to the formula. The process of adding a conflict-driven clause to the CNF is known as *clause learning* [MSS99, BS97, ZMMM01, BKS04]. A key question in this regard is when exactly to add this clause. Consider the termination tree in Figure 3.6 and the left most leaf node corresponding to the CNF $\Delta|A, B, C, X$. Unit resolution discovers a contradiction in this CNF and by analyzing the implication graph in Figure 3.7(a), we can identify the conflict-driven clause $\neg A \vee \neg X$. The question now is: What to do next?

Since the contradiction was discovered after setting variable X to **true**, we know that we have to at least undo that decision. Modern SAT solvers, however, will undo all decisions made after the *assertion level*, which is the second highest level in a conflict-driven clause. For example, in the clause $\neg A \vee \neg X$, A was set at Level 0 and X was set at level 3. Hence, the assertion level is 0 in this case. If the clause contains only literals from one level, its assertion level is then -1 by definition. The assertion level is special in the sense that it is the deepest level at which adding the conflict-driven clause would allow unit resolution to derive a new implication using that clause. This is the reason why modern SAT solvers would actually backtrack all the way to the assertion level, add the conflict-driven clause to the CNF, apply unit resolution, and then continue the search process. This particular method of performing non-chronological backtracking is referred to as *far-backtracking* [SBK05].

3.6.3.4. Clause Learning and Proof Complexity

The advantages of clause learning can also be shown from proof complexity perspective [CR79]. Whenever the CNF formula is unsatisfiable, a SAT solver can

Algorithm 3.5 DPLL+(CNF Δ): returns UNSATISFIABLE or SATISFIABLE.

```

1:  $D \leftarrow ()$  {empty decision sequence}
2:  $\Gamma \leftarrow \{\}$  {empty set of learned clauses}
3: while true do
4:   if unit resolution detects a contradiction in  $(\Delta, \Gamma, D)$  then
5:     if  $D = ()$  then {contradiction without any decisions}
6:       return UNSATISFIABLE
7:     else {backtrack to assertion level}
8:        $\alpha \leftarrow$  asserting clause
9:        $m \leftarrow$  assertion level of clause  $\alpha$ 
10:       $D \leftarrow$  first  $m$  decisions in  $D$  {erase decisions  $\ell_{m+1}, \dots$ }
11:      add clause  $\alpha$  to  $\Gamma$ 
12:   else {unit resolution does not detect a contradiction}
13:     if  $\ell$  is a literal where neither  $\ell$  nor  $\neg\ell$  are implied by unit resolution from  $(\Delta, \Gamma, D)$ 
14:       then
15:          $D \leftarrow D; \ell$  {add new decision to sequence  $D$ }
16:       else
17:         return SATISFIABLE

```

be thought of as a resolution proof engine that tries to produce a (short) proof for the unsatisfiability of the formula. From this viewpoint, the goal of the solver is to derive the empty clause. Generating short resolution proofs is difficult to implement in practice, because the right order of clauses to resolve is often hard to find. As a result, SAT solvers only apply a restricted version of resolution, which may generate longer proofs, to allow the solvers to run efficiently.

In [BKS04], the authors show that DPLL-based SAT solvers that utilize clause learning can generate exponentially shorter proofs than those that do not use clause learning. Moreover, it has been shown, in the same paper, that if the solver is allowed to branch on assigned variables and to use unlimited restarts, the resulting proof engine is theoretically as strong as an unrestricted resolution proof system.

3.6.4. Putting It All Together

Algorithm DPLL+ is the final SAT algorithm in this chapter which incorporates all techniques we discussed so far in this section. The algorithm maintains a triplet (Δ, Γ, D) consisting of the original CNF Δ , the set of learned clauses Γ and a decision sequence $D = (\ell_0, \dots, \ell_n)$, where ℓ_i is a literal representing a decision made at level i . Algorithm DPLL+ starts initially with an empty decision sequence D and an empty set of learned clauses Γ . It then keeps adding decisions to the sequence D until a contradiction is detected by unit resolution. If the contradiction is detected in the context of some decisions, a conflict-driven clause α is constructed and added to the set of learned clauses, while backtracking to the assertion level of clause α . The process keeps repeating until either a contradiction is detected without any decisions (unsatisfiable), or until every variable is set to a value without a contradiction (satisfiable). Part 1, Chapter 4 discusses in details the implementation of this type of SAT solvers.

3.6.4.1. An Example

To consider a concrete example of DPLL+, let us go back to the CNF 3.1 shown again below:

$$\Delta = \begin{array}{l} 1. \{A, B\} \\ 2. \{B, C\} \\ 3. \{\neg A, \neg X, Y\} \\ 4. \{\neg A, X, Z\} \\ 5. \{\neg A, \neg Y, Z\} \\ 6. \{\neg A, X, \neg Z\} \\ 7. \{\neg A, \neg Y, \neg Z\} \end{array}$$

DPLL+ starts with an empty decision sequence and an empty set of learned clauses:

$$\begin{array}{l} D = (), \\ \Gamma = \{ \}. \end{array}$$

Suppose now that the algorithm makes the following decisions in a row:

$$D = (A=\text{true}, B=\text{true}, C=\text{true}, X=\text{true}).$$

Unit resolution will not detect a contradiction until after the last decision $X=\text{true}$ has been made. This triggers a conflict analysis based on the implication graph in Figure 3.7(a), leading to the conflict-driven clause

$$8. \{\neg A, \neg X\},$$

whose assertion level is 0. Backtracking to the assertion level gives:

$$\begin{array}{l} D = (A=\text{true}), \\ \Gamma = \{\{\neg A, \neg X\}\}. \end{array}$$

Unit resolution will then detect another contradiction, leading to conflict analysis based on the implication graph in Figure 3.7(b). The conflict-driven clause in this case is

$$9. \{\neg A\}$$

with an assertion level of -1 . Backtracking leads to:

$$\begin{array}{l} D = (), \\ \Gamma = \{\{\neg A, \neg X\}, \{\neg A\}\}. \end{array}$$

No contradiction is detected by unit resolution at this point, so the algorithm proceeds to add a new decision and so on.

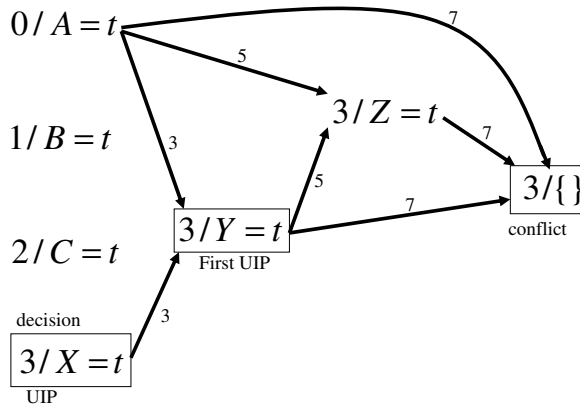


Figure 3.9. An example of a unique implication point (UIP).

3.6.4.2. More on Asserting Clauses

Modern SAT solvers insist on generating asserting conflict-driven clauses as they are guaranteed to become unit clauses when added to the CNF. Recall that a conflict-driven clause is asserting if it includes exactly one variable that has been set at the decision level where the contradiction is found. A popular refinement on this definition, however, appeals to the notion of *unique implication point (UIP)* [MSS99, ZMMM01]. In particular, a UIP of a decision level in an implication graph is a variable setting at that decision level which lies on every path from the decision variable of that level to the contradiction. Intuitively, a UIP of a level is an assignment at the level that, by itself, is sufficient for implying the contradiction. In Figure 3.9, the variable setting $3/Y=\text{true}$ and $3/X=\text{true}$ would be UIPs as they lie on every path from the decision $3/X=\text{true}$ to the contradiction $3/\{\}$.

Note that there may be more than one UIP for a given level and contradiction. In such a case, we will order the UIPs according to their distance to the contradiction node. The first UIP is the one closest to the contradiction. Even though there are many possible ways of generating asserting conflict-driven clauses from different UIPs, asserting clauses that contain the first UIP of the conflict level are preferred because they tend to be shorter.⁸ This scheme of deriving asserting clauses is called the *1UIP scheme*. One could also derive clauses that contain the first UIPs of other levels as well. However, the studies in [ZMMM01] and [DHN05b] showed that insisting on the first UIPs of other levels tends to worsen the solver's performance. Considering Figure 3.9, the conflict set $\{A=\text{true}, Y=\text{true}\}$ contains the first UIP of the conflict level (Y), leading to the asserting conflict-driven clause $\neg A \vee \neg Y$.

⁸A short clause is preferable because it potentially allows the solver to prune more search space.

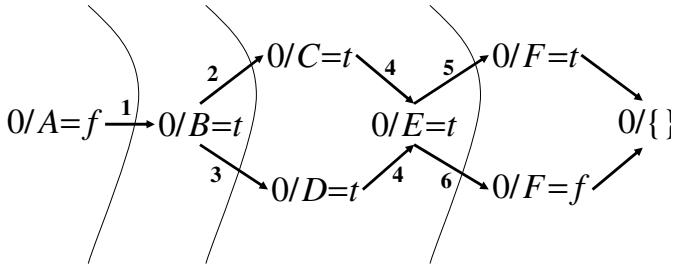


Figure 3.10. An implication graph with three different asserting, conflict-driven clauses.

3.6.4.3. Repeating Decision Sequences

Consider now the following CNF:

$$\Delta = \begin{array}{l} 1. \{A, B\} \\ 2. \{\neg B, C\} \\ 3. \{\neg B, D\} \\ 4. \{\neg C, \neg D, E\} \\ 5. \{\neg E, F\} \\ 6. \{\neg E, \neg F\} \end{array}$$

After DPLL+ has decided $\neg A$, unit resolution will detect a contradiction as given by the implication graph in Figure 3.10. There are three asserting clauses in this case, $\{A\}$, $\{\neg B\}$ and $\{\neg E\}$, where $\{\neg E\}$ is the first UIP clause. Suppose that DPLL+ generates $\{\neg E\}$. DPLL+ will then erase the decision $\neg A$ and add clause $\{\neg E\}$ to the formula. Unit resolution will not detect a contradiction in this case. Moreover, since neither A nor $\neg A$ is implied by unit resolution, DPLL+ may decide $\neg A$ again!

This example illustrates the extent to which DPLL+ deviates from the standard depth-first search used by DPLL. In particular, in such a search no decision sequence will be examined more than once as the search tree is traversed in a systematic fashion, ensuring that no node in the tree will be visited more than once. The situation is different with DPLL+ which does not have a memory of what decision sequences it has considered, leading it to possibly consider the same decision sequence more than once. This, however, should not affect the completeness of the algorithm, because the assignments at the time of future visits will be different. In this sense, even though the algorithm repeats the decision sequence, it does so in a different context. A completeness proof of DPLL+ is presented in [ZM03, Rya04].

3.6.4.4. Deleting Conflict-Driven Clauses

The addition of conflict-driven clauses is quite useful as it empowers unit resolution, allowing it to detect certain contradictions that it could not detect before. The addition of conflict-driven clauses may carry a disadvantage though, as it may considerably increase the size of given CNF, possibly causing a space problem. Moreover, the more clauses the CNF has, the more time it takes for the

solver to perform unit resolution. In practice, it is not uncommon for the number of added clauses to be much greater than the number of clauses in the original CNF on a difficult problem.

This issue can be dealt with in at least two ways. First, newly added conflict-driven clauses may actually subsume older conflict-driven clauses, which can be removed in this case. Second, one may decide to delete conflict-driven clauses if efficiency or space becomes an issue. In this case, the size, age, and activity of a conflict-driven clause are typically used in considering its deletion, with a preference towards deleting longer, older and less active clauses. Different conflict-driven clause deletion policies, mostly based on some heuristics, have been proposed in different SAT solvers [MSS99, MMZ⁺01, ES03].

3.6.4.5. Restarts

Another important technique employed by all modern SAT solvers in this category is *restarts* [GSC97]. When a SAT solver restarts, it abandons the current assignments and starts the search at the root of the search tree, while maintaining other information, notably conflict-directed clauses, obtained during earlier search. Restarting is a way of dealing with the heavy-tailed distribution of running time often found in combinatorial search [GSC97, BMS00]. Intuitively, restarting is meant to prevent the solver from getting stuck in a part of the search space that contains no solution. The solver can often get into such situation because some incorrect assignments were committed early on and unit resolution was unable to detect them. If the search space is large, it may take very long for these incorrect assignments to be fixed. Hence, in practice, SAT solvers usually restart after a certain number of conflicts is detected (indicating that the current search space is difficult), hoping that, with additional information, they can make better early assignments. The extensive study of different restarting policies in [Hua07b] shows that it is advantageous to restart often. The policy that was found to be empirically best in [Hua07b] is also becoming a standard practice in state-of-the-art SAT solvers.

Note that restarts may compromise the solver's completeness, when it is employed in a solver that periodically deletes conflict-driven clauses. Unless the solver allows the number of conflict-driven clauses to grow sufficiently large or eventually allows a sufficiently long period without restarting, it may keep exploring the same assignments after each restart. Note that both restarting and clause deletion strategies are further discussed in Part 1, Chapter 4.

3.6.5. Other Inference Techniques

Other than the standard techniques mentioned in the previous sub-sections, over the years, other inference techniques have been studied in the context of DPLL-based algorithm. Even though most of these techniques are not present in leading general complete SAT solvers, they could provide considerable speedup on certain families of problems that exhibit the right structure. We discuss some of these techniques in this brief section.

One natural extension of unit resolution is binary resolution. Binary resolution is a resolution in which at least one resolved clause has size two. Binary

resolution allows more implications to be realized at the expense of efficiency. In [VG01], Van Gelder presented a SAT solver, 2cIVER, that utilized a limited form of binary resolution. Bacchus also studied, in [Bac02], the same form of binary resolution, termed BinRes, in DPLL-based solver called 2CLS+EQ. This solver also employed other rules such as hyper resolution (resolution involving more than two clause at the same time) and equality reduction (replacing equivalent literals). The solver was shown to be competitive with zChaff [MMZ⁺01] (which only used unit resolution) on some families. Another solver that utilized equivalency reasoning was developed by Li [Li00]. The solver, called EqSatz, employed six inference rules aimed at deriving equivalence relations between literals and was shown to dominate other solvers on the DIMACS 32-bit parity problems [SKM97] (among others). SymChaff [Sab05], developed by Sabharwal, takes advantage of symmetry information (made explicit by the problem creator) to prune redundant parts of the search space to achieve exponential speedup.

3.6.6. Certifying SAT Algorithms

As satisfiability testing becomes vital for many applications, the need to ensure the correctness of the answers produced by SAT solving algorithms increases. In some applications, SAT algorithms are used to verify the correctness of hardware and software designs. Usually, unsatisfiability tells us that the design is free of certain types of bug. If the design is critical for safety or the success of the mission, it would be a good idea to check that the SAT solvers did not produce a buggy result.

Verifying that a SAT solving algorithm produces the right result for satisfiable instances is straightforward. We could require the SAT solving algorithm to output a satisfying assignment. The verifier then needs to check whether every clause in the instance mention at least a literal from the satisfying assignment. Verifying that a SAT solving algorithm produces the right result for unsatisfiable instances is more problematic, however. The complications are caused by the needs to make the verifying procedure simple and the “certificate” concise; see [Van02]. The main method used for showing unsatisfiability of a formula is to give a resolution proof of the empty clause from the original set of clauses.

One of the most basic certificate formats is the explicit resolution derivation proof. A certificate in this format contains the complete resolution proof of unsatisfiability (the empty clause). The proof is simply a sequence of resolutions. First of all, all variables and clauses in the original formula are indexed. Each resolution is made explicit by listing the indices of the two operands, the index of the variable to be resolved on, and the literals in the resolvent. This certificate format was referred to as %RES and was used for the verification track of the SAT’05 competition [BS05].

A closely related format called resolution proof trace (%RPT) [BS05] does not require the resolvent of each resolution to be materialized. Hence, the certificate will only list the clauses to be resolved for each resolution operation.

Zhang and Malik proposed in [ZM03] a less verbose variant which relies on the verifier’s ability to reconstruct implication graphs in order to derive learned clauses. According to this format, the certificate contains, for each learned clause

generated by the solving algorithm, the indices of all clauses that were used in the derivation of the conflict-driven clause. The verifier then has the burden of reconstructing the conflict-driven clauses that are required to arrive at the proof of the empty clause.

Another certificate format was proposed by Goldberg and Novikov in [GN03]. This format lists all conflict-driven clauses generated by the SAT solving algorithm in the order of their creation. This type of certificate employs the fact that if a learned clause C is derived from the CNF Δ , then applying unit resolution to $\Delta \wedge \neg C$ will result in a contradiction. By listing every conflict-driven clause (including the empty clause at the end) in order, the verifier may check to make sure that each conflict-driven clause can actually be derived from the set of preceding clauses (including the original formula) by using unit resolution.

3.7. Conclusions

We have discussed in this chapter various complete algorithms for SAT solving. We categorized these algorithms based on their approaches for achieving completeness: by existential quantification, by inference rules, and by search. While much work has been done in each category, nowadays, it is the search-based approach that receives the most attention because of its efficiency and versatility on many real-world problems. We presented in this chapter the standard techniques used by leading contemporary SAT solvers and some less common inference techniques studied by others. Current research in this area focuses on both theory and applications of satisfiability testing. Recently, however, there is a growing emphasis on applying a SAT solver as a general-purpose problem solver for different problem domains (see Part 3). As a result, a branch of current research focuses on finding efficient ways to encode real-world problems compactly as CNF formulas. When it comes to using a SAT solver as a general-purpose solver, efficiency is the most important issue. Consequently, techniques that are theoretically attractive may not necessarily be adopted in practice, unless they can be efficiently implemented and shown to work on a broad set of problems.

References

- [AMS01] F. A. Aloul, I. L. Markov, and K. A. Sakallah. Faster sat and smaller bdds via common function structure. In *Technical Report #CSE-TR-445-01*. University of Michigan, November 2001.
- [AV01] A. S. M. Aguirre and M. Y. Vardi. Random 3-SAT and BDDs: The plot thickens further. In *Principles and Practice of Constraint Programming*, pages 121–136, 2001.
- [Bac02] F. Bacchus. Enhancing davis putnam with extended binary clause reasoning. In *AAAI/IAAI*, pages 613–619, 2002.
- [BCM⁺90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

- [Bie07] A. Biere. Picosat versions 535, 2007. Solver description, SAT Competition 2007.
- [BKS04] P. Beame, H. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004.
- [BMS00] L. Baptista and J. P. Marques-Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In *Principles and Practice of Constraint Programming*, pages 489–494, 2000.
- [Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
- [BS97] R. J. J. Bayardo and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 203–208, Providence, Rhode Island, 1997.
- [BS05] D. L. Berre and L. Simon, 2005. SAT'05 Competition Homepage, <http://www.satcompetition.org/2005/>.
- [CR79] S. A. Cook and R. A. Reckhow. The relative efficiency of propositional proof systems. *J. Symb. Log.*, 44(1):36–50, 1979.
- [CS00] P. Chatalic and L. Simon. ZRes: The old Davis-Putnam procedure meets ZBDDs. In D. McAllester, editor, *17th International Conference on Automated Deduction (CADE'17)*, number 1831, pages 449–454, 2000.
- [Dar01] A. Darwiche. Recursive conditioning. *Artificial Intelligence*, 126(1-2):5–41, 2001.
- [Dav84] R. Davis. Diagnostic reasoning based on structure and behavior. *Artif. Intell.*, 24(1-3):347–410, 1984.
- [Dec86] R. Dechter. Learning while searching in constraint-satisfaction problems. In *AAAI*, pages 178–185, 1986.
- [Dec90] R. Dechter. Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. *Artif. Intell.*, 41(3):273–312, 1990.
- [Dec97] R. Dechter. Bucket elimination: A unifying framework for processing hard and soft constraints. *Constraints: An International Journal*, 2:51–55, 1997.
- [DHN05a] N. Dershowitz, Z. Hanna, and A. Nadel. A clause-based heuristic for sat solvers. In *Proceedings of SAT 2005*, pages 46–60, 2005.
- [DHN05b] N. Dershowitz, Z. Hanna, and A. Nadel. Towards a better understanding of the functionality of a conflict-driven sat solver. In *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing*, 2005.
- [dKW87] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artif. Intell.*, 32(1):97–130, 1987.
- [DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.
- [DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.
- [DR94] R. Dechter and I. Rish. Directional resolution: The davis putnam

- procedure, revisited. In *Principles of Knowledge Representation (KR-94)*, pages 134–145, 1994.
- [ES03] N. Eén and N. Sörensson. An extensible sat-solver. In *Proceedings of SAT'03*, pages 502–518, 2003.
- [FKS⁺04] J. Franco, M. Kouril, J. Schlipf, J. Ward, S. Weaver, M. Dransfield, and W. M. Vanfleet. Sbsat: a state-based, bdd-based satisfiability solver. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing*, pages 398–410, 2004.
- [Gas77] J. Gaschnig. A general backtrack algorithm that eliminates most redundant tests. In *IJCAI*, page 457, 1977.
- [Gen84] M. R. Genesereth. The use of design descriptions in automated diagnosis. *Artif. Intell.*, 24(1-3):411–436, 1984.
- [GN02] E. Goldberg and Y. Novikov. Berkmin: A fast and robust sat-solver. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 142, Washington, DC, USA, 2002. IEEE Computer Society.
- [GN03] E. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for cnf formulas. In *Proceedings of Design, Automation and Test in Europe (DATE2003)*., 2003.
- [GSC97] C. P. Gomes, B. Selman, and N. Crato. Heavy-tailed distributions in combinatorial search. In *Principles and Practice of Constraint Programming*, pages 121–135, 1997.
- [GW00] J. F. Groote and J. P. Warners. The propositional formula checker heerhugo. *Journal of Automated Reasoning*, 24:101–125, 2000.
- [Har96] J. Harrison. Stalmårck's algorithm as a hol derived rule. In *Proceedings of TPHOLS'96*, pages 221–234, 1996.
- [HD04] J. Huang and A. Darwiche. Toward good elimination orders for symbolic sat solving. In *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 566–573, 2004.
- [HKB96] R. Hojati, S. C. Krishnan, and R. K. Brayton. Early quantification and partitioned transition relations. In *ICCD '96: Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors*, pages 12–19, Washington, DC, USA, 1996. IEEE Computer Society.
- [Hua07a] J. Huang. A case for simple sat solvers. In *CP*, pages 839–846, 2007.
- [Hua07b] J. Huang. The effect of restarts on the efficiency of clause learning. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 2318–2323, 2007.
- [JS04] H. Jin and F. Somenzi. Circus : Hybrid satisfiability solver. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, 2004.
- [Li00] C. M. Li. Integrating equivalency reasoning into davis-putnam procedure. In *AAAI: 17th National Conference on Artificial Intelligence*. AAAI / MIT Press, 2000.
- [LLM03] J. Lang, P. Liberatore, and P. Marquis. Propositional independence: Formula-variable independence and forgetting. *J. Artif. Intell. Res.*

- (*JAIR*), 18:391–443, 2003.
- [Mar00] P. Marquis. Consequence finding algorithms. *Handbook on Defeasible Reasoning and Uncertainty Management Systems, Volume 5: Algorithms for Uncertain and Defeasible Reasoning*, pages 41–145, 2000.
- [MM02] D. Motter and I. Markov. A compressed breadth-first search for satisfiability. In *Proceedings of ALENEX 2002.*, 2002.
- [MMZ⁺01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. *39th Design Automation Conference (DAC)*, 2001.
- [MSS99] J. P. Marques-Silva and K. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, (5):506–521, 1999.
- [PD07] K. Pipatsrisawat and A. Darwiche. Rsat 2.0: Sat solver description. Technical Report D–153, Automated Reasoning Group, Computer Science Department, UCLA, 2007.
- [Pro93] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, (9(3)):268–299, August 1993.
- [PV04] G. Pan and M. Vardi. Symbolic techniques in satisfiability solving. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, 2004.
- [RAB⁺95] R. Ranjan, A. Aziz, R. Brayton, B. Plessier, and C. Pixley. Efficient bdd algorithms for fsm synthesis and verification. In *Proceedings of IEEE/ACM International Workshop on Logic Synthesis, Lake Tahoe, USA, May 1995.*, 1995.
- [Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.
- [Rya04] L. Ryan. Efficient Algorithms for Clause-Learning SAT Solvers. Master’s thesis, Simon Fraser University, 2004.
- [Sab05] A. Sabharwal. Symchaff: A structure-aware satisfiability solver. In *AAAI*, pages 467–474, 2005.
- [SB06] C. Sinz and A. Biere. Extended resolution proofs for conjoining bdds. In *CSR*, pages 600–611, 2006.
- [SBK05] T. Sang, P. Beame, and H. A. Kautz. Heuristics for fast exact model counting. In *Proceedings of SAT’05*, pages 226–240, 2005.
- [SKM97] B. Selman, H. A. Kautz, and D. A. McAllester. Ten challenges in propositional reasoning and search. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI’97)*, pages 50–54, 1997.
- [SS77] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9, October 1977.
- [SS90] G. Stålmarek and M. Säflund. Modeling and verifying systems and software in propositional logic. In *B.K. Daniels, editor, Safety of Computer Control Systems, 1990 (SAFECOMP’90)*, 1990.
- [Stå94] G. Stålmarek. System for determining propositional logic theorems by applying values and rules to triplets that are generated from

- boolean formula., 1994. United States Patent number 5,276,897; see also Swedish Patent 467 076.
- [TSL⁺90] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using bdd's. In *Proceedings of IEEE International Conference on Computer-Aided Design (ICCAD-90)*., pages 130–133, 1990.
- [TY84] R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3):566–579, 1984.
- [Van02] A. Van Gelder. Extracting (easily) checkable proofs from a satisfiability solver that employs both preorder and postorder resolution. In *AMAI*, 2002.
- [VG01] A. Van Gelder. Combining preorder and postorder resolution in a satisfiability solver. In *IEEE/ASL LICS Satisfiability Workshop*, Boston, 2001.
- [Wid96] F. Widebäck. Stålmärck's notion of n-saturation. Technical Report NP-K-FW-200, Prover Technology AB, 1996.
- [ZM03] L. Zhang and S. Malik. Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10880, Washington, DC, USA, 2003. IEEE Computer Society.
- [ZMMM01] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.

Chapter 4

Conflict-Driven Clause Learning SAT Solvers

Joao Marques-Silva, Ines Lynce, and Sharad Malik

4.1. Introduction

The decision problem for propositional logic, commonly referred to as the Boolean Satisfiability (SAT) problem, is well-known to be NP-complete [Coo71].¹ Although NP-completeness implies (under the reasonable assumption that $P \neq NP$) that any complete SAT algorithm runs in worst-case exponential time, it is also the case that SAT solvers most often defy this predicted worst-case behavior. Indeed, SAT can be viewed as a success story of computer science. From an academic curiosity in the early 90s, when solving a formula with a few hundred variables would be considered a significant success, SAT solvers have metamorphosed into highly efficient engines that routinely solve formulas with millions of variables, and which find widespread industrial deployment. Conflict-Driven Clause Learning (CDCL) SAT solving is the sole reason for this success.

The development of CDCL SAT solvers was from the outset motivated by practical applications. During the late 80s and early 90s, applications of SAT in automatic test pattern-generation (ATPG) [Lar89, Lar92] and timing analysis [MSS⁺91, SBSV96] motivated the development of clause learning and the GRASP SAT algorithm [MS95, MSS96c, MSS99]. Another development was the application of SAT to automated planning [KS92, KS96], but here the initial focus was on exploiting (incomplete) local-search SAT solvers, which reflected the strong prominence of incomplete SAT throughout the 90s [SLM92]. The initial practical success of GRASP [MSS96a] led to the inclusion of GRASP-like clause learning in SATO [Zha97], which in turn was featured in the original SAT-based model checking papers [BCCZ99, BCC⁺99]. The immediate impact of SAT-based model checking, and the performance improvements achieved with Chaff [MMZ⁺01, ZMMM01], resulted in a stream of improvements in SAT solvers, and later in SMT solvers [BSST09], which catalyzed the many applications we see today.

¹Besides the work of S. Cook, NP-completeness was investigated independently by L. Levin [Lev73].

The range of practical applications of CDCL SAT solving has continued to expand in recent years, arguably at an ever increasing pace. The practical significance of CDCL SAT solvers has also influenced the organization of a number of other automated reasoning engines, ranging from the propositional domain to the first-order logic setting, and also finding applications for reasoning about logics more expressive than first-order [GK03, Kor08, KV13, Vor14]. Moreover, CDCL SAT solvers have impacted research in several other areas. The applications of CDCL SAT solvers motivated the development of new propositional encodings [Wal00, Gen02, BB03, GN04, AM04, Sin05, ES06, Pre07, Pre09, RM09, BBR09, ANORC09, CZI10, ANORC11b, ANORC11a, ANO⁺12, OLH⁺13, Kar18, MJML14, MPS14]. The usage of CDCL SAT solvers as oracles (for the class NP) motivated extensive revisions of the algorithms for a number of problems, including maximum satisfiability [MSP07, MMSP09, ABL09, DB11, ABL13, IMM⁺14, MJML14, MDMS14, MHL⁺13, SBJ16], tackling enumeration problems [NBE12, JMSS14, PIMMS15, LPMMS16, PMJaMS18, GIL18], solving various quantification problems [JMS11, JKMSC12, RT15, JKMSC16, RS16, IPMS15, IJMS16, SWJ16, IMMS16, CLLB⁺17, LLBdLM18], and finding minimal sets [GMP08, MSJB13, MS10, Nad10, MSL11, BLMS12, MSHJ⁺13, NRS13, GLM14a, BDTK14, BK15, MPMS15, NBMS18], among others. Furthermore, CDCL SAT solvers have also led to a series of new results in propositional proof complexity, but also to the search for the automation of proof systems stronger than resolution [BKS04, PD09, AFT09, Hua10, AKS10, PD11, AFT11, IMMS17b, BBI⁺18, EN18, EGCNV18]. Given the above, it is reasonable to argue that CDCL SAT solving has become one of the mainstays of research in the general areas of constraint solving and optimization, computational logic and automated reasoning.

The purpose of this chapter is to provide an overview of the organization and practical uses of CDCL SAT solvers, but also to give an historical perspective to the key ideas that integrate CDCL SAT solvers. The chapter covers the organization of CDCL SAT solvers as instantiated in most modern tools.² Promising techniques, that have been integrated in a smaller number of solvers, will also be described. The approach taken is to describe the key ideas behind CDCL SAT solvers, providing a comprehensive range of examples that illustrate how these ideas work in practice. The chapter also provides a brief overview of the many uses of CDCL SAT solvers, some of which represent existing lines of research. For somewhat different treatments of the same subject, the interested reader is referred to a number of related overviews which have been published over the years [MS98, MSS00, ZM02, LMS02, LMS03, Mit05, CESS08, MS08, Bie08b, GKSS08, MSLM09, MZ09, Knu15, VWM15, MSM18].

The chapter is organized as follows. Section 4.2 introduces the notation used in the remainder of the chapter. Section 4.3 describes the implementation of CDCL SAT solvers. Section 4.4 outlines the ways in which CDCL SAT solvers are used in practical settings. Section 4.5 provides a glimpse of the practical and theoretical significance of CDCL SAT solving. Moreover, Section 4.6 pro-

²Compared to the first edition of the Handbook of Satisfiability, the chapter has been extensively rewritten, with the purpose of providing a more modern treatment of CDCL, taking into account relevant work over the last decade, but also aiming to provide a glimpse into the many practical successes of CDCL SAT solving.

vides a brief historical perspective on the development of CDCL SAT solvers. Section 4.7 provides additional pointers for examples of SAT solvers as well as the PySAT framework for prototyping with SAT solvers [IMMS18]. Finally, Section 4.8 summarizes the chapter but also research directions for the improvement and application of CDCL SAT solvers.

4.2. Preliminaries

This section introduces the notation and definitions used throughout. The goal is to describe variants of backtrack search algorithms, and so the definitions reflect this goal. Some notation is adapted from [BL99, BHvMW09].

A set \mathbb{V} of propositional variables is assumed. Variables will be lower case letters, with or without indices, e.g. $\{a, b, c, \dots, x_1, x_2, \dots, y_1, y_2, \dots\}$. Propositional formulas are defined inductively over \mathbb{V} , using the standard logical connectives \neg , \vee and \wedge , as follows:

1. $\mathcal{F} = x$, with $x \in \mathbb{V}$, is a propositional formula.
2. If \mathcal{F} is a propositional formula, then $(\neg\mathcal{F})$ is a propositional formula.
3. If \mathcal{F} and \mathcal{G} are propositional formulas, then $(\mathcal{F} \vee \mathcal{G})$ is a propositional formula.
4. If \mathcal{F} and \mathcal{G} are propositional formulas, then $(\mathcal{F} \wedge \mathcal{G})$ is a propositional formula.

The definition can be extended to consider other logical connectives.

Most SAT solvers operate on the more restricted representation of Conjunctive Normal Form (CNF) formulas. Each clause c_i is defined as a set of literals. A literal is either a variable $x \in \mathbb{V}$ or its complement \bar{x} .³ We also use $var(l_i)$ to denote the variable associated with some literal l_i . A set of clauses will be interpreted as a conjunction of clauses. A clause will be interpreted as a disjunction of literals. Throughout this chapter, both representations will be used.

Example 4.2.1 (CNF Formula). An example of a CNF formula is:

$$\mathcal{F} = \{\{a, \bar{b}\}, \{b, c\}, \{b, \bar{d}\}, \{\bar{a}, \bar{c}, d\}\} \quad (4.1)$$

The alternative clausal representation is:

$$\mathcal{F} = (a \vee \bar{b}) \wedge (b \vee c) \wedge (b \vee \bar{d}) \wedge (\bar{a} \vee \bar{c} \vee d) \quad (4.2)$$

Propositional variables can be assigned a propositional value, taken from $\{0, 1\}$. However, in the context of search algorithms for SAT, variables may also be *unassigned*. As a result, assignments to propositional variables are defined as a function $\nu : \mathbb{V} \rightarrow \{0, u, 1\}$, where u denotes an *undefined* value used when a variable has not been assigned a value in $\{0, 1\}$. (ν could also be defined as a partial map from \mathbb{V} to $\{0, 1\}$, but we opt to use the additional value u .) Given an assignment ν , if all variables are assigned a value in $\{0, 1\}$, then ν is referred to as a *complete assignment*. Otherwise it is a *partial assignment*.

Given a truth assignment ν , the value taken by a propositional formula, denoted \mathcal{F}^ν , is defined inductively as follows:

³To keep the notation as simple as possible, we will represent negative literals by \bar{x} instead of the often used notation $\neg x$.

1. If $\mathcal{F} = x$, with $x \in \mathbb{V}$, then $\mathcal{F}^\nu = \nu(x)$.
2. If $\mathcal{F} = (\neg\mathcal{G})$, then

$$\mathcal{F}^\nu = \begin{cases} 0 & \text{if } \mathcal{G}^\nu = 1 \\ 1 & \text{if } \mathcal{G}^\nu = 0 \\ u & \text{otherwise} \end{cases}$$

3. If $\mathcal{F} = (\mathcal{E} \vee \mathcal{G})$, then

$$\mathcal{F}^\nu = \begin{cases} 1 & \text{if } \mathcal{E}^\nu = 1 \text{ or } \mathcal{G}^\nu = 1 \\ 0 & \text{if } \mathcal{E}^\nu = 0 \text{ and } \mathcal{G}^\nu = 0 \\ u & \text{otherwise} \end{cases}$$

4. If $\mathcal{F} = (\mathcal{E} \wedge \mathcal{G})$, then

$$\mathcal{F}^\nu = \begin{cases} 1 & \text{if } \mathcal{E}^\nu = 1 \text{ and } \mathcal{G}^\nu = 1 \\ 0 & \text{if } \mathcal{E}^\nu = 0 \text{ or } \mathcal{G}^\nu = 0 \\ u & \text{otherwise} \end{cases}$$

The definition of value of a formula applies to arbitrary propositional formulas, and so also to CNF formulas. An assignment that satisfies a propositional formula is referred to as a *model* (or as a *satisfying assignment*).

The assignment function ν will also be viewed as a set of tuples (x_i, v_i) , with $v_i \in \{0, 1\}$. Adding a tuple (x_i, v_i) to ν corresponds to assigning v_i to x_i , such that $\nu(x_i) = v_i$. Removing a tuple (x_i, v_i) from ν , with $\nu(x_i) \neq u$, corresponds to assigning u to x_i . With a mild abuse of notation, we use $x_i = v_i$ when $\nu(x_i) = v_i$ and ν is clear from the context. Moreover, variable assignments will also be represented as (true) literals. Concretely, if x_i is used, then it indicates that $\nu(x_i) = 1$, whereas if \bar{x}_i is used, then it indicates that $\nu(x_i) = 0$.

Clauses are characterized as *falsified*, *satisfied*, *unit* or *unresolved*. A clause is falsified if all its literals are assigned value 0. A clause is satisfied if at least one of its literals is assigned value 1. A clause is unit if all literals but one are assigned value 0, and the remaining literal is unassigned. Finally, a clause is unresolved if it is neither unsatisfied, nor satisfied, nor unit.

A key procedure in SAT solvers is the *unit clause rule* [DP60]: if a clause is unit, then its sole unassigned literal must be assigned value 1 for the clause to be satisfied. The iterated application of the unit clause rule is referred to as *unit propagation* or *Boolean constraint propagation* (BCP) [ZM88]. In modern CDCL solvers, as in most implementations of the DPLL algorithm [DP60, DLL62], logical consequences are derived with unit propagation. Unit propagation is applied after each branching step but also during preprocessing,⁴ and is used for identifying variables which must be assigned a specific Boolean value. If an unsatisfied clause is identified, a *conflict* condition is declared, and the algorithm needs to backtrack, and may implement some sort of conflict analysis procedure.

Example 4.2.2. Consider the following formula:

$$\begin{aligned} \mathcal{F} &= \mathbf{c}_1 \wedge \mathbf{c}_2 \wedge \mathbf{c}_3 \wedge \mathbf{c}_4 \\ &= (a) \wedge (\bar{a} \vee b) \wedge (\bar{a} \vee c) \wedge (\bar{b} \vee \bar{c} \vee d) \end{aligned}$$

⁴A branching step is a choice of a variable and a propositional value to assign to the variable.

The operation of unit propagation in this example works as follows. Clause \mathbf{c}_1 is unit. As a result, a must be assigned value 1. Moreover, since $a = 1$, then the clauses \mathbf{c}_2 and \mathbf{c}_3 are also unit (since the literal $\bar{a} = 0$). Thus, the assignments $b = 1$ and $c = 1$ are necessary. Finally, because of these two assignments, then clause \mathbf{c}_4 is also unit, and so the assignment $d = 1$ is also necessary.

In CDCL SAT solvers, each variable x_i is characterized by a number of properties, including the *value*, the *antecedent* (or *reason*) and the *decision level*, denoted respectively by $\nu(x_i) \in \{\mathbf{0}, \mathbf{u}, 1\}$ (introduced above), $\alpha(x_i) \in \mathcal{F} \cup \{\mathbf{n}, \mathfrak{d}\}$ (where \mathbf{n}, \mathfrak{d} are special symbols introduced below), and $\delta(x_i) \in \{\mathbf{u}, 0, 1, \dots, |X|\}$. A variable x_i that is assigned a value as the result of applying the unit clause rule is said to be *implied*. The unit clause \mathbf{c}_j used for implying variable x_i is said to be the antecedent of x_i , $\alpha(x_i) = \mathbf{c}_j$. For variables that are decision variables the antecedent is \mathfrak{d} and the decision level is the depth of the search tree associated with that decision variable. For unassigned variables, the antecedent is set to \mathbf{n} and the decision level is set to \mathbf{u} . Hence, antecedents are only assigned a clause identifier for variables whose value is implied by other assignments. Literals inherit decision level and antecedent information from the corresponding variable. For assigned variables, the value of a literal x_i is given by the value of variable x_i and the value of a literal \bar{x}_i is given by $1 - \nu(x_i)$. For unassigned variables, the value of any literal is given by $\nu(x_i) = \nu(\bar{x}_i) = \mathbf{u}$.

The decision level of a literal l_i , $\delta(l_i)$ is defined as the decision level of corresponding variable x_i , $\delta(x_i)$, if $l_i = x_i$ or $l_i = \neg x_i$. The decision level of a variable x_i (either if implied or if it is a decision variable) denotes the depth of the decision tree at which the variable is assigned a value in $\{0, 1\}$. The decision level for an unassigned variable x_i is \mathbf{u} . Moreover, $\delta(x_i) = \mathbf{u}$ when $\alpha(x_i) = \mathbf{n}$. The decision level associated with variables used for branching steps (i.e. *decision assignments*) is specified by the search process, and denotes the current depth of the *decision stack*. Hence, a variable x_i associated with a decision assignment is characterized by having $\alpha(x_i) = \mathfrak{d}$, $\delta(x_i) > 0$, and $\nu(x_i) \neq \mathbf{u}$. For a variable that is not a decision assignment, the decision level is defined as follows:

$$\delta(l_i) = \begin{cases} \mathbf{u} & \text{if } \alpha(l_i) = \mathbf{n} \\ \max(\{\mathbf{0}\} \cup \{\delta(l_j) \mid l_j \in \text{lits}(\alpha(l_i)) \setminus \{l_i\}\}) & \text{otherwise} \end{cases} \quad (4.3)$$

i.e. the decision level of an implied literal is either the highest decision level of the implied literals in a non-unit clause, or it is 0 if the antecedent is unit, i.e. there are no literals in $\alpha(l_i)$ other than l_i . The notation $x_i = v@d$ is used to denote that $\nu(x_i) = v$ and $\delta(x_i) = d$. Finally, the notation $@d$ may be used just to identify the decision level of some anonymous literal.

Example 4.2.3 (Decision Levels & Antecedents). Consider the CNF formula:

$$\begin{aligned} \mathcal{F} &= \mathbf{c}_1 \wedge \mathbf{c}_2 \wedge \mathbf{c}_3 \\ &= (x_1 \vee \bar{x}_4) \wedge (x_1 \vee x_3) \wedge (\bar{x}_3 \vee x_2 \vee x_4) \end{aligned}$$

Assume that the decision assignment is $x_4 = 0@1$. Unit propagation yields no additional implied assignments. Assume the second decision is $x_1 = 0@2$. Unit propagation yields the implied assignments $x_3 = 1@2$ and $x_2 = 1@2$. Moreover, $\alpha(x_3) = \mathbf{c}_2$ and $\alpha(x_2) = \mathbf{c}_3$.

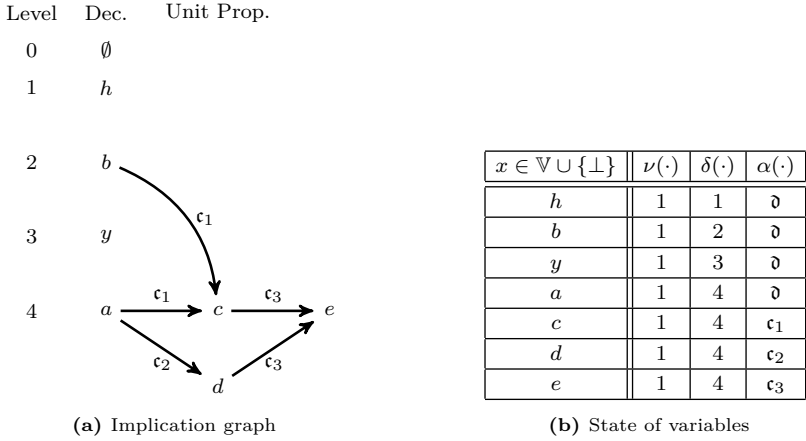


Figure 4.1. Implication graph for example 4.2.4

During the execution of a search-based SAT solver (which will be detailed later in this chapter), assigned variables as well as their antecedents define a directed acyclic graph $I = (V_I, E_I)$, referred to as the *implication graph* [MS95, MSS96c].

The vertices in the implication graph are defined by all assigned variables and one special node \perp , $V_I \subseteq \mathbb{V} \cup \{\perp\}$. The edges in the implication graph are obtained from the antecedent of each assigned variable: if $c_j = \alpha(x_i)$, then there is a directed edge from each variable in c_j , other than x_i , to x_i . If unit propagation yields an falsified clause c_j , then a special vertex \perp is used to represent the unsatisfied clause. In this case, the antecedent of \perp is defined by $\alpha(\perp) = c_j$.

Example 4.2.4 (Implication Graph without Conflict). Consider the CNF formula:

$$\begin{aligned} \mathcal{F}_1 &= c_1 \wedge c_2 \wedge c_3 \\ &= (\bar{a} \vee \bar{b} \vee c) \wedge (\bar{a} \vee d) \wedge (\bar{c} \vee \bar{d} \vee e) \end{aligned} \quad (4.4)$$

Assume the decision assignment $b = 1@2$. Moreover, assume that the current decision assignment is $a = 1@4$. The resulting implication graph is shown in figure 4.1.⁵

Example 4.2.5 (Implication Graph with Conflict). Consider the CNF formula:

$$\begin{aligned} \mathcal{F}_2 &= c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge c_5 \wedge c_6 \\ &= (\bar{a} \vee \bar{b} \vee c) \wedge (\bar{a} \vee d) \wedge (\bar{c} \vee \bar{d} \vee e) \wedge (\bar{h} \vee \bar{e} \vee f) \wedge (\bar{e} \vee g) \wedge (\bar{f} \vee \bar{g}) \end{aligned} \quad (4.5)$$

Assume decision assignments $h = 1@1$, $b = 1@2$ and $y = 1@3$. Moreover, assume that the current decision assignment is $a = 0@4$. The resulting implication graph, that takes into account unit propagation at decision level 4, is shown in figure 4.2, and yields a conflict because clause $(\bar{f} \vee \bar{g})$ becomes falsified.

⁵To keep the notation as simple as possible, all variables are assigned value 1 in the examples, either as decision assignments or as implied assignments. A few exceptions will be noted.

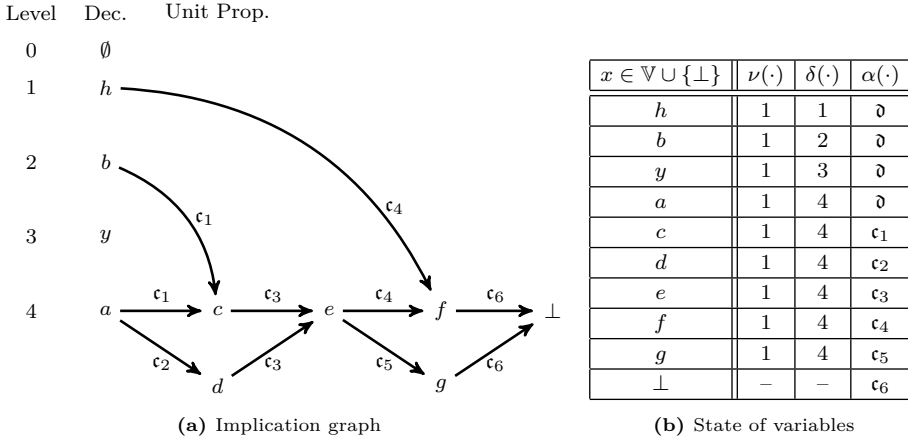


Figure 4.2. Implication graph for example 4.2.5

```

Input : CNF Formula  $\mathcal{F}$ 
Output: Outcome  $st$ 
DPLL ( $\mathcal{F}$ )
  DLevel  $\leftarrow$  0
  if not UnitPropagation() then
    return false
  while not AllVariablesAssigned() do
    DLevel  $\leftarrow$  DLevel + 1
    (var, val)  $\leftarrow$  PickBranchVariable()
    Assign(var, val, DLevel,  $\bar{d}$ )
    while not UnitPropagation() do
      BLevel  $\leftarrow$  NextUntoggledDecision()
      if BLevel == 0 then
        return false
      Backtrack(BLevel)
      DLevel  $\leftarrow$  BLevel
      (var, val)  $\leftarrow$  ToggleDecision(DLevel)
      Assign(var, val, DLevel,  $\bar{d}$ )
  return true
    
```

Algorithm 1: The DPLL Algorithm

It should be observed that the notation introduced above is relevant for describing the operation of a modern CDCL SAT solver. A DPLL-like SAT solver usually manipulates significantly less information.

The high-level organization of DPLL [DLL62] is shown in Algorithm 1.

Finally, the original justification for the DPLL algorithm was the propositional resolution operation [DP60]:⁶

$$\frac{x \vee \alpha \quad \bar{x} \vee \beta}{\alpha \vee \beta} \tag{4.6}$$

⁶Robinson’s well-known resolution principle for first-order logic was proposed at a later date [Rob65].

Resolution is a complete decision procedure for CNF formulas [DP60], but impractical in practice. The DPLL algorithm represented an attempt at devising a practically effective solution based on resolution [DLL62]. For historical reasons, the acronym DPLL (Davis-Putnam-Logemann-Loveland) reflects the origins of the DPLL algorithm.

4.3. Implementing CDCL SAT Solvers

This section describes the organization of CDCL SAT solvers, starting from the implementation of DPLL outlined in the previous section. The section is broadly organized into two main parts. The first part covers the CDCL techniques that are commonly employed by most modern CDCL SAT solvers. The second part covers other less used techniques.

4.3.1. Organization of a CDCL SAT Solver

The CDCL SAT algorithm shares similarities with the standard backtracking search procedure also used in the DPLL algorithm, where unit propagation is executed after each decision assignment (which creates a new decision level). However, a few fundamental differences exist. First, backtracking can be non-chronological and be related with learning clauses from conflicts [MS95, MSS96c, MSS99]. Second, backtracking occurs after every conflict [MMZ⁺01, ZMMM01]. Third, the algorithm may decide to *restart* the search from the first decision level [GSC97, GSK98, GSCK00, BMS00]. Furthermore, there are a few additional differences with respect to DPLL, in that learned clauses may eventually be deleted, and highly efficient (lazy) data structures are used. The top-level organization of the CDCL algorithm is shown in Algorithm 2. The following sections detail the differences of CDCL-like algorithms with respect to the DPLL algorithm.

4.3.1.1. Clause Learning

This section provides an overview of one of the hallmarks of CDCL SAT solvers, namely *clause learning*. During backtrack search, clauses are learned after falsified clauses are identified, using a procedure referred to as *conflict analysis* [MS95, MSS96c, MSS96b, MSS96a, MSS99].

4.3.1.1.1. Conflict Analysis. Given a falsified clause, conflict analysis iteratively traces antecedents of variables assigned at the current decision level, and records dependencies of variables assigned at smaller decision levels. The algorithm operates as follows. Variables assigned at the current decision level are analyzed in a first-in first-out fashion (thus being manipulated through a queue), ensuring that a variable is analyzed only after all variables it implies have been analyzed. At each step, the head of the queue is extracted, and its antecedent analyzed. Any literals assigned at decision levels smaller than the current one are added to (i.e. recorded in) the clause being learned. The antecedent tracing procedure terminates when the decision variable for the current decision level is visited. Clearly,

```

Input : CNF Formula  $\mathcal{F}$ 
Output: Outcome  $st$ 

CDCL ( $\mathcal{F}$ )
  DLevel  $\leftarrow$  0
  if not UnitPropagation() then
    return false
  while not AllVariablesAssigned() do
    DLevel  $\leftarrow$  DLevel + 1
    (var, val)  $\leftarrow$  PickBranchVariable()
    Assign(var, val, DLevel,  $\varnothing$ )
    while not UnitPropagation() do
      if DLevel == 0 then
        return false
      BLevel  $\leftarrow$  ConflictAnalysis()
      Backtrack(BLevel)
      DLevel  $\leftarrow$  BLevel
      if TimeToRestart() then
        Backtrack(0)
        DLevel  $\leftarrow$  0
  return true

```

Algorithm 2: The CDCL Algorithm**Table 4.1.** Execution of conflict analysis procedure

Step	Var Queue	Extract Var	Antecedent	Recorded Lits	Added to Queue
0	–	\perp	c_6	\emptyset	$\{f, g\}$
1	$[f, g]$	f	c_4	$\{\bar{h}\}$	$\{e\}$
2	$[g, e]$	g	c_5	$\{\bar{h}\}$	\emptyset
3	$[e]$	e	c_3	$\{\bar{h}\}$	$\{c, d\}$
4	$[c, d]$	c	c_1	$\{\bar{h}, \bar{b}\}$	$\{a\}$
4	$[d, a]$	d	c_2	$\{\bar{h}, \bar{b}\}$	\emptyset
5	$[a]$	a	\varnothing	$\{\bar{h}, \bar{b}, \bar{a}\}$	–
6	$[]$	–	–	$\{\bar{h}, \bar{b}, \bar{a}\}$	–

the conflict analysis procedure runs in worst-case linear time in the number of literals.

Example 4.3.1. For the implication graph from Example 4.2.5, the conflict analysis procedure is illustrated in Table 4.1. At step 0, the initialization step, the falsified clause (c_6) is analyzed and its variables, f and g , both assigned at decision level 4, are added to the queue of variables to visit. At step 1, the top of the queue is (for example) f . The antecedent of f is c_4 . As a result, the literal \bar{h} is recorded, and the variable e is added to the queue. The same process is repeated for steps 2 to 6. As can be observed, all variables assigned at the current decision level are analyzed in order. Any recorded literals are assigned at decision levels smaller than the current one. Hence, in this case, the learned clause is $(\bar{h} \vee \bar{b} \vee \bar{a})$. For this example, the variables and antecedents that are traced are highlighted in Figure 4.3.

Although CDCL SAT solvers implement the conflict analysis procedure as

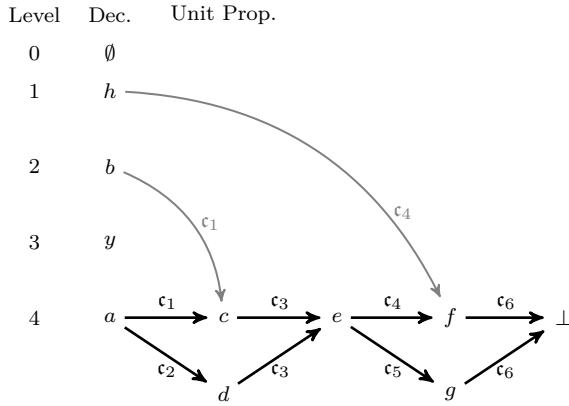


Figure 4.3. Traced antecedents during conflict analysis

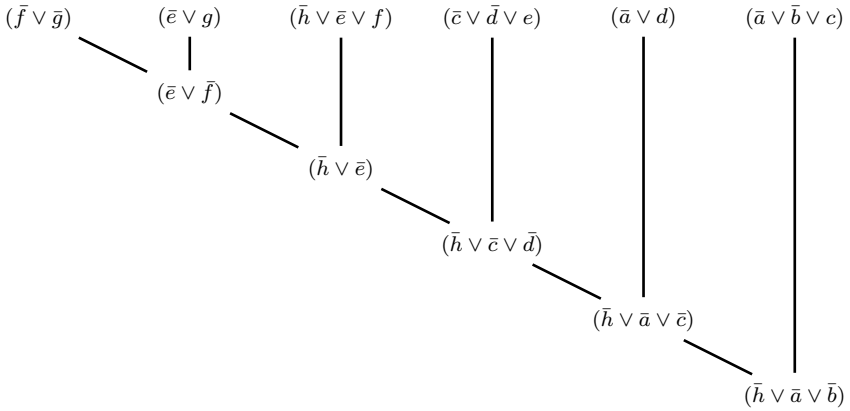


Figure 4.4. Explaining a learned clause with resolution steps

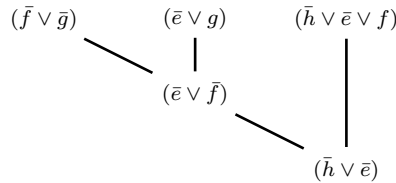
outlined above, there is a tight relationship with resolution, in that learned clauses can be explained with sequences of resolution steps. Concretely, each learned clause obtained from an implication sequence can be explained by *linear input regular resolution* [VG11] (also referred to as trivial resolution steps [BKS04]). As a result, there is a direct mapping between the conflict analysis procedure and linear input regular resolution steps.

Example 4.3.2. Regarding Example 4.3.1, the sequence of resolution steps that explains the learned clause $(\bar{h} \vee \bar{b} \vee \bar{a})$ is shown in Figure 4.4.

A key observation is that although conflict analysis can be explained with resolution steps, to our best knowledge there exists no other approach for automating resolution that is as effective as the conflict analysis procedure devised for CDCL SAT solvers. Even the DPLL algorithm [DLL62, DP60] corresponds to a weaker proof system, i.e. tree-like resolution [BKS04].

Table 4.2. First UIP clause learning

Step	Var Queue	Extract Var	Antecedent	Recorded Lits	Added to Queue
0	–	\perp	c_6	\emptyset	$\{f, g\}$
1	$[f, g]$	f	c_4	$\{\bar{h}\}$	$\{e\}$
2	$[g, e]$	g	c_5	$\{\bar{h}\}$	\emptyset
3	$[e]$	e	c_3	$\{\bar{h}, \bar{e}\}$	\emptyset
6	\square	–	–	$\{\bar{h}, \bar{e}\}$	–

**Figure 4.5.** Resolution steps with first UIP learning

4.3.1.1.2. Unique Implication Points. The basic clause learning algorithm can be further optimized by taking into account the *structure* of the conflicting implication sequence. As can be observed in the execution of the clause learning procedure in Table 4.1, in step 3 there exists only one variable to trace, concretely e . This variable is referred to as a *unique implication point* (UIP) [MS95, MSS96a, MSS96b, MSS96c, MSS99]. UIPs exhibit a number of important properties. For the implication graph restricted to variables assigned at the current decision level, a UIP is a *dominator* of the decision variable with respect to the conflict node \perp . The same assignment to any UIP guarantees that the same conflict will be reproduced. In modern CDCL SAT solvers, clause learning stops at the first UIP [MMZ⁺01, ZMMM01]. Nevertheless, conflict analysis need not stop at the first UIP, and initial CDCL SAT solvers would learn a clause at each UIP [MS95, MSS96a, MSS99]. This capability of conflict analysis to learn multiple clauses given UIPs is not available in most modern CDCL SAT solvers, despite recent promising results [SSS12].

Example 4.3.3. Regarding Example 4.2.5, with basic conflict analysis shown in Example 4.3.1, e is a UIP. In modern CDCL SAT solvers, the conflict analysis algorithm terminates when this first UIP is reached. In this case, the algorithm learns the clause $(\bar{h} \vee \bar{e})$. The sequence of resolution steps is shown in Figure 4.5. It should be noted that, if conflict analysis were to consider more UIPs, the clause $(\bar{a} \vee \bar{b} \vee e)$ would be learned for UIP a , the decision variable.

4.3.1.1.3. Non-Chronological Backtracking. Learned clauses enable SAT solvers to perform non-chronological backtracking [MS95, MSS96c, MSS99]. A wide range of modern CDCL SAT solvers backtrack (often non-chronologically) after each conflict. This is shown in Algorithm 2. The backtracking decision level is defined as the largest decision level of the literals that remain assigned after conflict analysis. It is immediate to conclude that clause learning at the first UIP

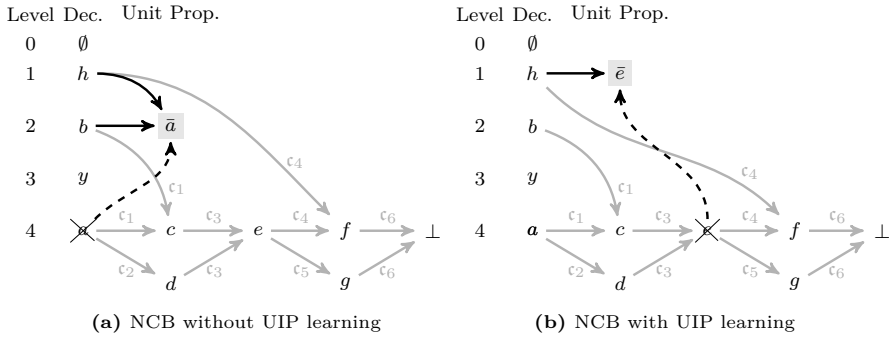


Figure 4.6. Non-chronological backtracking (NCB)

yields the most aggressive backtracking decision level [ABH⁺08]. Indeed, extending clause learning to any other UIP can only increase the number of traced literals, and so can only maintain or increase the backtracking decision level.

Starting with the SAT solver Chaff, the non-chronological backtracking step would be taken after each conflict [MMZ⁺01]. This is in contrast with the backtracking strategy used in the GRASP SAT solver [MS95, MSS96c, MSS99], where the non-chronological backtracking step would be taken *only* after the two assignments to a given decision variable had been considered. The backtracking strategy used in GRASP, improved with heuristics to decide when to apply it, has become popular in recent SAT solvers [NR18, MB19], being referred to as a *chronological backtracking* step when implementing *non-chronological backtracking*.

Example 4.3.4. For Example 4.3.1, the learned clause is $(\bar{h} \vee \bar{b} \vee \bar{a})$. After conflict analysis, at decision level 4, the clause becomes unit, due to a becoming unassigned. The largest decision level is 2, and so the CDCL algorithm backtracks to decision level 2, where the learned clause is still unit and implies the assignment $a = 0$. In case conflict analysis stops at the first UIP (see Example 4.3.3), then the learned clause is $(\bar{h} \vee \bar{e})$, the CDCL algorithm backtracks to decision level 1, and the learned clause implies the assignment $e = 0$. As can also be observed from the first UIP clause, exploiting UIPs enables more aggressive backtracking, in this case to decision level 1. Backtracking with and without UIP learning is shown in Figure 4.6.

4.3.1.1.4. Learned Clause Minimization. Although the conflict analysis procedure has remained unchanged since it was first proposed [MS95, MSS96c, MSS99], improvements have been made on how to make the learned clauses more effective in practice, concretely by filtering literals in the learned clause that can be deemed irrelevant. This section overviews two approaches for removing redundant literals from a learned clause [SB09]. One approach exploits the concept of self-subsuming resolution [SP04]. The other approach traces antecedents at decision levels lower than the current one, aiming at discovering subsets of literals in the learned clause that are implied by the other literals.

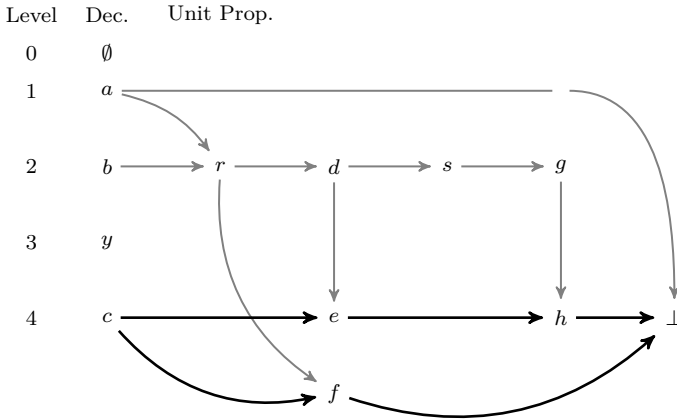


Figure 4.7. Clause minimization example

The first approach for learned clause minimization, referred to as *local minimization*, consists of applying self-subsumption resolution⁷ among the literals in the learned clause, aiming at identifying literals that can be dropped from the learned clause.

A more sophisticated (and more time-consuming) procedure, referred to as *recursive minimization*, traverses the implication graph at decision levels *smaller* than the current one:

1. Given a learned clause, pick some candidate literal, and mark all the other literals.
2. Recursively trace antecedents, starting from candidate, until marked literals or decision variables are reached.
3. If only marked literals are visited, then candidate can be declared redundant.
4. Repeat for all other literals in clause (except for decision literals).

Some practical implementations approximate exact recursive minimization [SB09]. Moreover, existing implementations ensure that the amortized run time is linear in the size of the implication graph (and so linear on the number of variables). Additional improvements to conflict clause minimization have been proposed in recent years [VG09, LLX⁺17].

The relationship of local and recursive learned clause minimization with resolution is immediate. Both clause minimization schemes identify additional resolution steps which enable the learned clause to be simpler.

Example 4.3.5. Consider the following CNF formula:

$$\begin{aligned}
 \mathcal{F} &= c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge c_5 \wedge c_6 \wedge c_7 \wedge c_8 \\
 &= (\bar{a} \vee \bar{b} \vee r) \wedge (\bar{r} \vee d) \wedge (\bar{c} \vee \bar{d} \vee e) \wedge (\bar{c} \vee \bar{d} \vee f) \wedge (\bar{d} \vee s) \wedge \\
 &\quad (\bar{s} \vee g) \wedge (\bar{e} \vee \bar{g} \vee h) \wedge (\bar{f} \vee \bar{h} \vee \bar{a})
 \end{aligned}$$

⁷Given $(x \vee \alpha)$ and $(\bar{x} \vee \alpha \vee \beta)$, self-subsuming resolution produces the clause $(\alpha \vee \beta)$.

Table 4.3. Execution of recursive minimization

Target	Curr Var	Marked	Unmarked	Vars to Trace	Action
g	g	$\{a, d, r, c\}$	\emptyset	$[s]$	–
g	s	$\{a, d, r, c\}$	\emptyset	$[d]$	–
g	d	$\{a, d, r, c\}$	\emptyset	\square	d marked, skip
g	–	$\{a, d, r, c\}$	\emptyset	\square	no unmarked vars; \therefore drop g
d	d	$\{a, r, c\}$	\emptyset	$[r]$	–
d	r	$\{a, r, c\}$	\emptyset	\square	r marked, skip
d	–	$\{a, r, c\}$	\emptyset	\square	no unmarked vars; \therefore drop d
r	r	$\{a, c\}$	\emptyset	$[a, b]$	–
r	a	$\{a, c\}$	\emptyset	$[c]$	a marked
r	b	$\{a, c\}$	$\{b\}$	\square	b decision & unmarked
r	–	$\{a, c\}$	$\{b\}$	\square	unmarked vars; \therefore keep r
a, c	–	–	\emptyset	\square	a, c decision variables; keep both

Consider the sequence of decisions: $\langle a = 1, b = 1, y = 1, c = 1 \rangle$ and the resulting implication graph, as shown in Figure 4.7. Conflict analysis yields the (first UIP) clause: $(\bar{a} \vee \bar{g} \vee \bar{d} \vee \bar{r} \vee \bar{c})$. Local minimization exploits the fact that self-subsuming resolution between the learned clause and \mathbf{c}_2 reduces the number of literals by one, thus producing: $(\bar{a} \vee \bar{g} \vee \bar{r} \vee \bar{c})$. Recursive minimization enables one further simplification, producing the final minimized clause: $(\bar{a} \vee \bar{r} \vee \bar{c})$. Alternatively, the sole execution of recursive minimization is summarized in Table 4.3. The literal on variable g is dropped because tracing only reaches variable d , which is marked (as it is included in the learned clause). The same applies to d . The literal on variable r cannot be dropped, since tracing reaches b , which is not marked. The literals of variables a and c cannot be dropped, since these are decision variables. Moreover, Figure 4.8 summarizes clause learning and also local and recursive minimization. Figure 4.8a, Figure 4.8b and Figure 4.8c show, respectively, the resolution derivation for the original learned clause, the result with local minimization, and the result with recursive minimization.

4.3.1.2. Search Restarts

In the late 90s, researchers observed that *branching randomization* would cause DPLL-like SAT solvers to exhibit heavy-tailed behavior on satisfiable formulas [GSC97, GSK98]. Concretely, the run time of a DPLL-like SAT solver on some satisfiable formulas would be small with high probability, but also the probability mass corresponding to large run times would be non-negligible, i.e. the run time distribution on satisfiable instances exhibited a heavy tail [GSC97, GSK98, GSCK00, GS09]. Given this behavior, a natural solution for improving the run time of DPLL-like SAT solvers on satisfiable instances would be to exploit branching randomization and apply a policy to restart the search often, e.g. after a number of conflicts.

Example 4.3.6 (Search Restarts). The example sketched in Figure 4.9 illustrates the execution of restarts. After a cutoff, the search is restarted. For completeness,

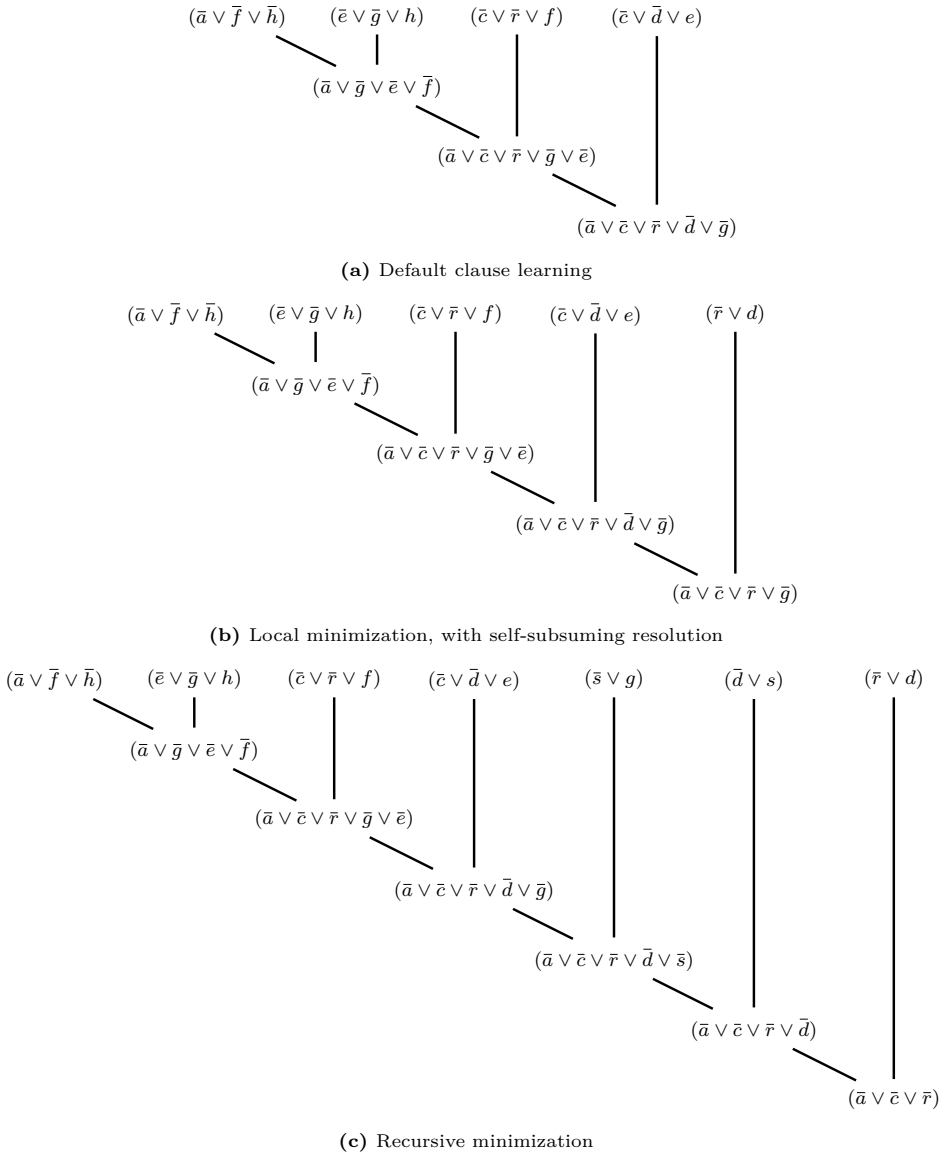


Figure 4.8. Learned clause minimization for Example 4.3.5

an increase in cutoff can be used.

The idea of search restarts was shown to work effectively with clause learning [BMS00, LBMS01], and eventually became an integral part of modern CDCL SAT solvers [MMZ⁺01, ES03, Bie08b]. More importantly, and from a theoretical perspective, it has been shown that clause learning and search restarts correspond to a proof system as powerful as general resolution, and stronger than the original

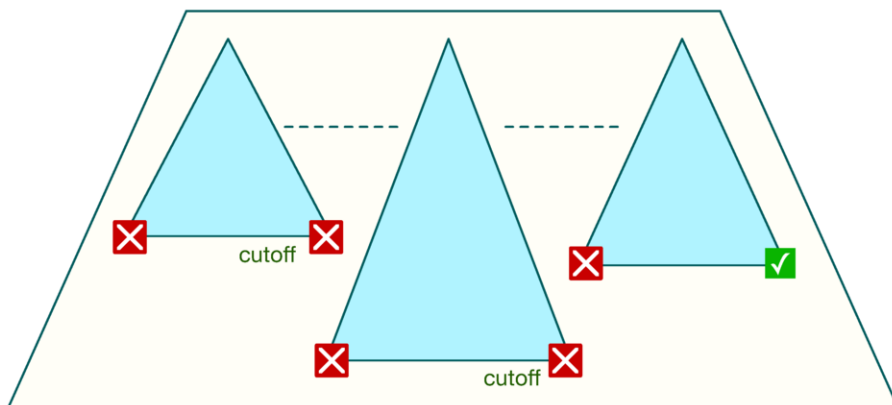


Figure 4.9. Operation of search restarts

DPLL proof system [PD09, PD11]. From a more practical perspective, branching randomization is still available in most modern CDCL SAT solvers, but mostly not used.

4.3.1.2.1. Selecting a Restart Sequence. In recent years, a number of authors proposed and evaluated different restart sequences [Hua07, Bie08b, Bie08a, SI09, BF15a]. One example are the so-called *Luby restarts* due to Luby, Sinclair and Zuckerman [LSZ93], and proposed in the CDCL SAT solver TiniSAT [Hua07]. In more recent work, different restart strategies are mixed, attempting to exploit different solver behavior between satisfiable and unsatisfiable instances [Oh15].

4.3.1.2.2. Ensuring Search Completeness. A possible limitation of search restarts is solver completeness. For example, always restarting after a fixed number of conflicts will yield an incomplete algorithm, which may be unable to solve some instances. This limitation was acknowledged in earlier work [GSK98, BMS00], and different solutions proposed. A simple solution is to increase the number of conflicts in between restarts [GSK98, BMS00]. Another solution is to ensure that enough learned clauses are kept in between restarts [LBMS01].

4.3.1.2.3. Phase Saving. A more recent technique is to keep the branching phase after the search is restarted [PD07]. Most modern CDCL SAT solvers implement the phase saving technique [Bie08a, AS09].

4.3.1.3. Deletion of Learned Clauses

From the outset [MS95, MSS96c], clause learning was proposed under the assumption that not all learned clauses would be kept, and that some of these clauses would eventually be deleted. With the improvements in performance from the early 00s [MMZ⁺01], policies for deciding which clauses to delete became paramount [GN02]. Clauses are deleted on a regular basis after some number of conflicts. The state of the art is to use the Literal Block Distance (LBD) clause quality metric to decide which clauses to delete [AS09]. Given a learned clause,

its literals are partitioned into decision levels. The number of decision levels is the LBD metric. Clauses with higher LBD metric are less likely to be relevant for future conflicts, and so can be considered for deletion. Moreover, the LBD metric has also been applied for deciding when to minimize clauses (since clauses with small LBD metric are more likely not to be deleted) [Bie13], and to decide the restart schedule [AS12, BF15a].

4.3.1.4. Conflict-Driven Branching

For a satisfiable instance, perfect branching would enable finding a satisfying assignment without conflicts, e.g. it would suffice to pick the variables and assignments from an existing model (if it were somehow known). Unfortunately, many instances are unsatisfiable and, even for the satisfiable instances, it seems fairly unlikely that a solver would be able to find the perfect way to branch.

Similarly to clause learning, which is only applied in the presence of conflicts, modern branching mechanisms aim to promote conflicts. The best known branching strategy is VSIDS (Variable State Independent Decaying Sum) [MMZ⁺01]. Over the years variants have been proposed [GN02], but all similar in spirit to VSIDS (and this is one critical contribution to the success of CDCL SAT solving). (See [BF15b] for a fairly recent account of branching heuristics.) The original VSIDS branching strategy can be summarized as follows:

1. A counter is associated with each variable.
2. After a clause is learned, the counters of the variables used for deriving the clause are incremented.
3. Branching consists of picking the unassigned variable with the highest counter, with ties broken randomly.
4. Counters are divided by a constant on a regular basis.

4.3.1.5. Lazy Data Structures

One additional ingredient for modern CDCL SAT solvers are lazy data structures. Modern CDCL SAT solvers can learn many clauses with a large number of literals. For these clauses, the classical data structure, associating a reference from each variable to each clause containing a literal in that variable, was effectively too slow to manage [MSS96c, MSS99]. An elegant solution is to consider lazy data structures, that do not maintain so many references [ZS00]. The most significant of these solutions is the watched literals data structure [MMZ⁺01].

Example 4.3.7 (Watched Literals). The example in Figure 4.10 will be used as a running example to describe the operation of watched literals. The clause c_t contains eight literals, named A, B, . . . , H, such that literal C is assigned value 0 at decision level 0, literal H is assigned value 0 at decision level 1, and literals B and D are assigned value 0 at decision level 2. (In general, shaded boxes denote assigned 0-valued literals.) After unit propagation at decision level 2, this clause c_t is unresolved with four literals still unassigned, concretely A, D, F and G.

The watched literals data structure is organized as follows [MMZ⁺01]:

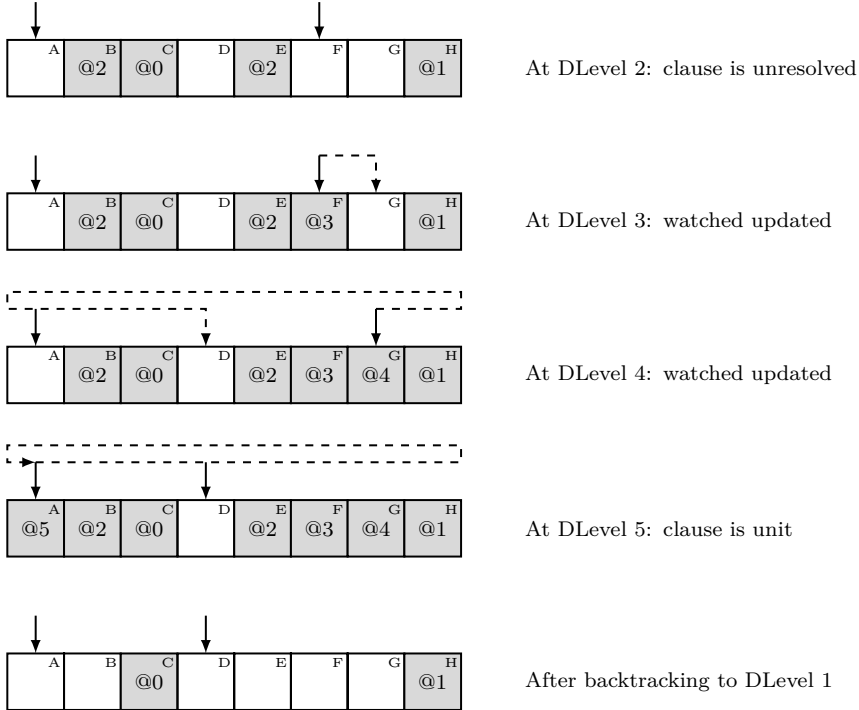


Figure 4.10. Watched literals

1. For each clause, *exactly* two unassigned literals are watched, using two *watch references*. For the running example, at decision level 2, the watch references point to the literals A and F.
2. The clause's state is only updated when one of the watched literals is assigned. If the watched literal is satisfied, nothing more is done with respect to this clause. If the literal is falsified, then the clause is traversed in search for one non-false literal that is not already being watched. For the running example, at decision level 3, since the literal in F is assigned, then the unassigned literal in G becomes watched. At decision level 4, it is necessary to traverse most of the clause's literals to conclude that the next literal to watch is D. Finally, at decision level 5, the literal in A becomes assigned. This results in all of the clause's literals to be traversed in search for another non-watched unassigned literal. In this case, there is none, and so all literals must be traversed, and the clause is declared unit.
3. When backtracking, the watched literals information is *not* updated. The ability to leave watch references unchanged when backtracking is arguably one of the most significant features of the watched literal data structure.

When a clause becomes satisfied the simplest solution is to leave the watch references untouched. It is possible to conceive mechanisms for *parking* the watch

references, such that these become unparked when the clause is no longer satisfied. Moreover, as Example 4.3.7 hints at, at each decision level, the worst-case amortized run time of updating the watched literals in a clause is linear in the clause's size. This is true if the watch references operate in circular mode (as shown in Example 4.3.7) and it is optimal [Gen13]. Some other ways of implementing watched literals (e.g. in MiniSat [ES03]) are known not to be optimal.

Some solvers implement dedicated data structures for specific types of clauses. For example, binary clauses can be represented as direct implications, which reduces not only the memory footprint, but also the overhead involved. Moreover, in many practical settings, binary clauses account for a significant percentage of the total number of clauses.

4.3.2. Additional Techniques

A number of promising techniques have been under active development in recent years, but do not yet find widespread use. An ongoing line of work, of key importance in some settings is preprocessing of CNF formulas and, more recently inprocessing [JHB12], which includes among others the blocked clause elimination technique [JBH10]. Additional recent efforts include exploiting machine learning techniques to optimize the branching heuristic [LGPC16a, LGPC16b] and more recently the restart strategy [LOM⁺18], an improved learned clause minimization procedure [LLX⁺17], and the use in some settings of GRASP's non-chronological backtracking strategy [NR18, MB19]. Finally, some solvers [SNC09, MHB12, Bie13] implement gaussian elimination [WvM98], with the purpose of more efficient handling of XOR constraints.

Additional low-level optimizations are often introduced in CDCL SAT solvers. The interested reader should read the available solver descriptions, e.g. from the SAT competitions [BBJS15, HS15, BJJ16, BJJ17a, BJJ17b, HJS18].

Given that problem representations are most often non-clausal [Stu13], one might expect that non-clausal CDCL SAT solvers would represent a promising direction for improving practical SAT solving. A number of efforts have been reported over the years [GeSSMS99, KGP01, MSGeS03, TBW04, DJN09]. However, the performance of non-clausal solvers has not kept up with that of clausal SAT solvers. Thus, the most commonly used solution is simply to clausify the problem representation. A similar situation was observed for other dedicated solvers operating over propositional domains [RM09].

A number of additional concepts have been the subject of a significant body of work, but such efforts have not materialized in their application in CDCL SAT solvers. Concrete examples include symmetry breaking [CGLR96, Sak09], backdoors and backbones [WGS03, KSTW05], but also phase transitions [MZK⁺99]. Backbones denote assignments that are fixed in any model of satisfiable formulas, and find a wide range of applications. However, in practice backbones are computed with CDCL SAT solvers, and have not served to improve the performance of CDCL SAT solvers. Backdoors represent sets of variables which, for some assignment, enable a polynomial solver to satisfy the formula in polynomial time. For unsatisfiable formulas, strong backdoors are sets of variables such that, for any assignment, unsatisfiability is decided in polynomial time. Although backdoors caused significant interest, in practice backdoors can be hard to compute.

Moreover, recent results [JJ09] cast doubts in the practical effectiveness of exploiting backdoors. To our best knowledge, CDCL SAT solvers are yet to exploit backdoors. Nevertheless, recent work exploits SAT backdoors for devising cryptographic attacks [SZO⁺18]. Finally, symmetries in SAT have been the subject of a comprehensive body of research for the last two decades. Important progress has been registered, but the performance gains of breaking symmetries can vary wildly. As a result, modern CDCL SAT solvers do not readily integrate identification of symmetry breaking predicates. Finally, phase transitions are a well-known phenomenon of combinatorial problems, denoting a transition from easy to hard problems on random instances and given some problem measure [MZK⁺99]. Insights from phase transitions are yet to be exploited by CDCL SAT solvers for improving run times of practically-oriented formulas.

4.4. Using CDCL SAT Solvers

The practical success of CDCL SAT solvers builds upon not only a highly efficient reasoning engine, but also upon a number of supporting technologies which have witnessed very significant progress, to a large extent motivated by the success of CDCL SAT solvers. These technologies include the many ways SAT solvers have been embedded in applications, but also in a wide range of reasoners, the many ways SAT solvers are used as oracles for the class NP of decision problems, and also, the many ways in which problems can be encoded into propositional domains. This section provides a glimpse of the approaches that have been devised for using SAT solvers in practical settings. The section does not aim to be exhaustive, but instead to provide a comprehensive list of pointers from which a better understanding of the range of uses of CDCL SAT solvers can be attained.

4.4.1. Practical Use of SAT Solvers

A number of practical successes of SAT solvers depend not only on the raw performance of the solvers, but also on a number of additional features that are available in most modern implementations of CDCL SAT solvers. In many settings, SAT solvers are iteratively used as (extended) oracles for the class NP of decision problems.

4.4.1.1. CDCL SAT vs. NP Oracles

The ideal model of an NP oracle provides an answer of Yes (or No) depending on whether some decision problem has (or does not have) a solution. When using SAT solvers as oracles for the class NP, there are important differences. First, SAT solvers compute a solution (i.e. the satisfying assignment) when the answer is positive. Abstracting the actual run time, (CDCL) SAT oracles can be modeled as witness-producing oracles for NP [JaMS16]. Moreover, for negative answers, SAT solvers are often capable of providing an *unsatisfiable core*, i.e. a subset of the original set of clauses, which is by itself unsatisfiable. SAT solvers offer no guarantee in terms of the quality of the computed unsatisfiable core, e.g. one possible unsatisfiable core is the formula itself. Moreover, SAT solvers provide no guarantee in terms of the quality of the resolution proof associated

with the computed unsatisfiable core. Despite this lack of guarantees in terms of the information that can be obtained given a negative outcome, and as shown in Section 4.4.3, CDCL SAT solvers find an ever increasing range of applications where both positive and negative outcomes are to be expected.

4.4.1.2. Incremental SAT with Assumptions

Most CDCL SAT solvers can be invoked given a number of assumed values on a set of variables; these are the so-called *assumptions*. In turn, this allows for incremental iterative use. Consider a clause c_i . If one needs to add or remove c_i in between calls to a SAT solver, then the clause that is actually added to the SAT solver data structures is: $(c_i \vee \bar{s}_i)$, where s_i denotes a *fresh variable* (referred to as *activation*, *selection* or *indicator* variable). As a result, a user can conditionally *program* the activation of clause c_i by setting an assumption $s_i = 1$ if the clause is to be considered, or $s_i = 0$ if the clause is not to be considered. Assumptions are passed to the SAT solver, assigned and propagated before the actual search starts. CDCL SAT solvers usually assign each one of these assumption variables per decision level, and put the reference decision level immediately after all assumptions have been handled. If the SAT solver backtracks to one of these decision levels and a conflict is identified, then the SAT solver returns unsatisfiable. A simple solution to delete clauses that can be incrementally activated/deactivated is to add a unit clause (\bar{s}_i) . Similarly, if the clause is to be declared final, meaning that it is to be used in all future calls to the SAT solver, but needs no longer to be activated/deactivated, then the solution is to add the unit clause (s_i) .

Example 4.4.1. Consider a SAT solver working with clauses taken from two sets, \mathcal{B} and \mathcal{S} :

$$\begin{aligned}\mathcal{B} &= \{(\bar{a} \vee b), (\bar{a} \vee c)\} \\ \mathcal{S} &= \{(a \vee \bar{s}_1), (\bar{b} \vee \bar{c} \vee \bar{s}_2), (a \vee \bar{c} \vee \bar{s}_3), (a \vee \bar{b} \vee \bar{s}_4)\}\end{aligned}$$

\mathcal{B} denotes background knowledge. These clauses are *final* and their activation *cannot* be controlled with indicator variables. In contrast, \mathcal{S} denotes clauses whose activation can be conditionally selected with the indicator variables $\{s_1, s_2, s_3, s_4\}$. It can also be observed that if all clauses in \mathcal{S} are activated, then the resulting formula is unsatisfiable. Nevertheless, given the set of assumptions $\{s_1 = 1, s_2 = 0, s_3 = 0, s_4 = 1\}$, the SAT solver effectively needs to decide the satisfiability of the following set of clauses:

$$\mathcal{F} = \{(\bar{a} \vee b), (\bar{a} \vee c), (a), (a \vee \bar{b})\}$$

It is immediate to conclude that \mathcal{F} is satisfiable with the satisfying assignment $\{a = 1, b = 1, c = 1\}$.

Incremental SAT solving using assumptions was first proposed in the MINISAT SAT solver [ES03], and since the mid 00s it has been applied in most applications where CDCL SAT solvers are used as (witness-producing) oracles. Incremental SAT solving with assumptions has enabled solving a growing number of problems, for which the SAT solver is used as an oracle. Section 4.4.3 provides

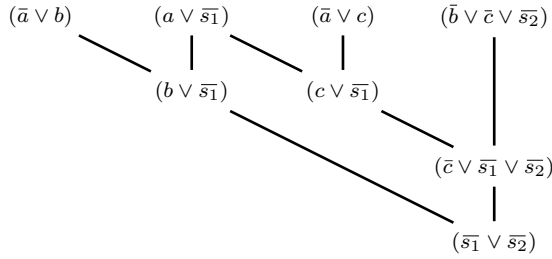


Figure 4.11. Finding unsatisfiable cores

a glimpse of the many applications where incremental SAT with assumption has been applied over the last decade. Moreover, a fairly recent topic of research is on improving the use of assumptions in such settings [LB13, ALS13].

Although the incremental SAT approach outlined in this section is standard, there have been recent alternative approaches to implement incremental SAT [NR12, NRS14, FBS19].

4.4.1.3. Unsatisfiable Core Extraction

In a number of practical settings one is often interested in understanding the causes of unsatisfiability. An often used representation of the causes of unsatisfiability are *unsatisfiable cores* (or unsatisfiable subsets), which denote subsets of a CNF formula which are by themselves unsatisfiable. For CDCL SAT solvers with an incremental interface, unsatisfiable cores can be computed by checking the assumptions that remain in the final learned clause.

For unsatisfiable formulas, assuming no indicator variables are in use, a SAT solver must eventually learn the empty clause, as the final confirmation that the formula is unsatisfiable. If indicator variables are in use there are a few important differences to note. Concretely, any indicator variable is necessarily transferred to a learned clause. As a result, for unsatisfiable formulas, instead of learning the empty clause, the SAT solver terminates by learning a clause that contains *all* the indicator variables of clauses used for proving (by resolution) the unsatisfiability of the formula. Clearly, it is easy to argue that the indicator variables in the final learned clause *must* be a superset of the clauses necessary to prove unsatisfiability by resolution.

Example 4.4.2. Consider again the sets of clauses \mathcal{B} and \mathcal{S} from Example 4.4.1. Moreover, given the assumptions $\{s_1 = 1, s_2 = 1, s_3 = 1, s_4 = 1\}$, an unsatisfiable core obtained by a CDCL SAT solver is represented by the indicator variables $\{s_1, s_2\}$. This signifies that the clauses associated with this set of indicator variables are inconsistent with the background knowledge. An explanation for the operation of the CDCL SAT solver is shown in Figure 4.11.

4.4.1.4. Proof Tracing

Applications of SAT solvers in safety-critical applications, or in computer-assisted proofs, motivate the need for SAT solvers to have their results validated. The val-

validation of positive (i.e. satisfiable) outcomes is straightforward. (This is a natural consequence of SAT being in the class NP, and so having succinct certificates.) Indeed, for a positive outcome, it suffices to validate the satisfying assignment against the CNF formula, which can be done in linear time on the size of the formula. In contrast, deciding unsatisfiability is in coNP, and so the existence of succinct certificates is unlikely. Interest in validating unsatisfiable outcomes can be traced at least to the early 00s [VG02], with the approaches now in common use first outlined in 2003 [ZM03, GN03]. In these approaches, CDCL SAT solvers are instrumented to output a trace of their execution, which can then be checked by validating the associated resolution proof. Existing approaches for checking proofs of unsatisfiability can rely on ad-hoc implementations (referred to as *proof checking*) or on implementations extracted from formally verified checkers (referred to as *proof certification*). A wealth of proof tracing formats have been proposed over the years for checking unsatisfiability proofs [Bie08b, Van08, Van12, HHW13, HHJW13, WHHJ14, HHJW14, CFMSSK17, CFHHJ⁺17], which evolved to enabling proof systems stronger than resolution, but also with recent formats aiming at efficient proof certification. Work on devising proof certification approaches has also witnessed remarkable progress over the last decade [Web06, WA09, DFMS10, WHHJ13, CFMSSK17, CFHHJ⁺17, HHJKW17].

The importance of checking (and of certifying) proofs of unsatisfiability cannot be overstated. For example, in recent years, solvers participating in the SAT competition are required to output traces in the case of unsatisfiable outcomes, to enable proof checking (and certification) [BHJ17a]. Furthermore, proof checking and certification has been at the core of computer-assisted mathematical proofs [KL14, KL15, HKM16, HK17, Heu18].

4.4.1.5. Warm Restarts

For iterative uses of SAT solvers, when the run time of each call to a SAT solver can be expected to be negligible, and in settings where the search tree has many levels, one technique to reduce the run time is to consider *warm restarts*⁸. The idea is to bootstrap the SAT solver with the final set of decisions from the last (positive) call to the SAT solver [JLSS14, PIMMS15, Pre15].

4.4.2. Modeling with Propositional Logic

The success of CDCL SAT solvers also motivated the quick growth of problem solving by encoding problems into propositional logic. Over the last two decades, a large number of novel propositional encodings have been proposed, which are used in an ever increasing range of applications. This section briefly overviews recent work on modeling with propositional logic.

Propositional logic is a fairly low level knowledge representation formalism. The following sections provide brief accounts of the approaches used to represent complex constraints with propositional logic. Well-known examples include cardinality constraints, pseudo-Boolean constraints, but also solutions for devising propositional encodings for more expressive domains.

⁸To our best knowledge, the term was coined by A. Biere around 2012.

For constraints defined on propositional variables, the most often used measure of quality of propositional encodings is whether arc-consistency in the original constraint can be achieved by unit propagation [Gen02, ES06, RM09]. When this is the case, the propositional encoding is said to *ensure* arc-consistency. The arc-consistency measure of quality is tightly related with the ability of unit propagation to reveal inconsistent assignment as early as possible. For constraints defined on finite domains a often used measure of quality is propagation completeness [BKN⁺09, BMS12].

4.4.2.1. Linear Inequalities

Linear inequalities over propositional variables are of the form:

$$\sum_{i=1}^n a_i x_i \bowtie b \quad (4.7)$$

where $\bowtie \in \{<, \leq, =, \geq, >\}$, and where in the general case $a_i, b \in \mathbb{R}$. A well-known encoding of linear inequalities into propositional logic was proposed by Warners [War98]. This encoding is linear on the size of the input representation (which can be exponential on the number of variables), but does not ensure arc-consistency. In SAT solving, linear inequalities are often organized into cardinality constraints and pseudo-Boolean constraints. For each, there are encodings which ensure arc-consistency, with worst-case polynomial representations on the size of the input representation. There are a few additional kinds of linear inequalities studied in recent years. The next sections briefly overview the different kinds of linear inequalities and their encodings. More detailed accounts can be found in standard references [Pre09, RM09].

4.4.2.1.1. Encoding Cardinality Constraints. One of the most often used types of constraints are cardinality constraints of the form:

$$\sum_{i=1}^n x_i \bowtie k \quad (4.8)$$

where $\bowtie \in \{<, \leq, =, \geq, >\}$. A well-known special case is $k = 1$, with specific often used constraints being referred to as *AtMost1*, *AtLeast1* and *Equals1*. Clearly, the case *AtLeast1* can be encoded with a single clause, and so the interesting case is the encoding of *AtMost1* constraints.

For the general case of cardinality constraints, some of the best known encodings include:

1. Sorting networks [Bat68, ES06].
2. Totalizers [BB03].
3. Cardinality networks [ANORC09, ANORC11b].
4. Pairwise cardinality networks [CZI10].
5. Modulo totalizer [OLH⁺13].

One can also consider a simple generalization of the pairwise encoding for *AtMost1* constraints, but the encoding size is exponential in n and k . There is also recent work on encodings that ensure arc-consistency [Kar18].

In addition to the encodings above, well-known encodings for the AtMost1 constraint include:

1. Ladder/regular [GN04, AM04].
2. Bitwise [FP01, Pre07].
3. And the straightforward pairwise encoding [Pre09].

All of the above encodings ensure arc-consistency.

4.4.2.1.2. Conditional & Soft Cardinality Constraints. In some settings cardinality constraints may be conditional, i.e. of the form:

$$y_j \rightarrow \left(\sum_i x_i \bowtie k \right) \quad (4.9)$$

depending on the value of y_j , the cardinality must (or needs not) hold. Conditional cardinality have been proposed and used in different settings, including to represent soft cardinality constraints [MDMS14] in MaxSAT, in bounded path finding [EN15], and for computing the max clique in undirected graphs [IMMS17a]. Recent work investigates mechanisms to ensure arc-consistency of conditional cardinality constraints [BJRS18].

4.4.2.1.3. Encoding Pseudo-Boolean Constraints. The most general form of linear inequalities, i.e. the so-called Pseudo-Boolean (PB) constraints can be framed as follows [RM09]:

$$\sum_{i=1}^n a_i l_i \geq b \quad (4.10)$$

where $l_i \in \{x_i, \bar{x}_i\}$, $x_i \in \{0, 1\}$, $a_i, b \in \mathbb{N}_0$.

Motivated by the applications of CDCL SAT solvers, and problem encodings into SAT, there has been recent work on finding encodings for PB constraints which ensure arc-consistency. Concrete examples include:

1. Binary decision diagrams (BDDs) [ES06], with the caveat of requiring a worst-case exponential representation on the size of the PB constraint representation.
2. Polynomial watchdog encoding [BBR09] which, to our best knowledge, was the first polynomial encoding of PB constraints that ensures arc-consistency.
3. Recent improvement to the polynomial watchdog encoding [ANORC11a, ANO+12, MPS14].

Besides the operational encoding of Warners [War98], other encodings have been proposed which do not ensure arc-consistency [ES06, MDMS14]. Similar to soft cardinality constraints, weighted boolean optimization [MMSP09] proposed the use of soft PB constraints.

4.4.2.2. Encoding More Expressive Domains

There has been extensive work on translating more expressive domains into SAT, to be solved with CDCL SAT solvers. Two well-known examples are planning

and (hardware) model checking [KS99, BCCZ99, BCC⁺99, Rin09]. Another well-known example is CSP [Wa00, Gen02], with a number of tools implementing translations of CSP into SAT. There have been efforts to encode problems in NP to SAT, e.g. NP-Spec [CS05]. Other examples include the family of relational model finders [Jac00, TJ07], but also encodings of programming languages into SAT [CKL04, Kro09].

4.4.3. SAT Solvers as Witness-Producing Oracles

The last decade has witness the quick growth of problem solving where oracles for NP are replaced with CDCL SAT oracles. As mentioned earlier, CDCL SAT solvers extend the notion of NP oracle by producing witnesses in the case of positive outcomes, and (compact) explanations (i.e. unsatisfiable cores) in the case of negative outcomes [JaMS16]. This section overviews this ongoing work. The section aims at illustrating the range of ideas being developed, but not at being exhaustive, both in the topics and in the references. The interested reader can find additional references in the references cited.

4.4.3.1. Finding Cardinality-Minimal Subsets

The identification of cardinality-minimal subsets finds many practical applications. One well-known example is Maximum Satisfiability (MaxSAT), where the goal is to compute a cardinality maximal set of constraints that is consistent. The problem is of interest when one starts from an inconsistent (i.e. unsatisfiable, often referred to as overconstrained) formula. The practical importance of MaxSAT is illustrated by recent surveys [MHL⁺13, SZGN17], covering a wide range of applications. Initial practical algorithms for MaxSAT focused on branch-and-bound search [LM09]. This state of affairs changed in 2006, with the FM core-guided algorithm for MaxSAT [FM06]. Core-guided MaxSAT algorithms are based on iteratively calling a CDCL SAT solver, finding unsatisfiable cores, and using relaxation variables and cardinality constraints for finding the least cost way of relaxing the CNF formula so as to recover consistency. Since the FM algorithm, a wealth of core-guided MaxSAT algorithms have been devised [MSP07, MMSP09, ABL09, ABL13, IMM⁺14, MJML14, MDMS14]. These include the now widely used MSU3 algorithm [MSP07] and variants [ABL13, MJML14], improvements to the based FM algorithms [MMSP09, ABL09], and the use of soft-cardinality constraints [MDMS14]. A more detailed account of earlier core-guided MaxSAT algorithms is included in a recent survey [MHL⁺13]. A different MaxSAT approach that exploits CDCL SAT solvers consists in the iterative computation of minimum hitting sets [DB11, SBJ16]. The computed minimum hitting sets identify a set of clauses to exclude. The rest is checked for satisfiability with a (CDCL) SAT solver. If the formula is satisfiable, then the result is the MaxSAT solution, otherwise a computed unsatisfiable core is added to the set of sets for which a minimum hitting set is to be computed, and the process is repeated.

Another example of computing cardinality-minimal subsets is minimal satisfiability (MinSAT) [KKM94]. As with MaxSAT, initial algorithms focused on branch-and-bound search [LMQZ10]. The use of CDCL SAT solvers, as oracles for

NP, for this problem was proposed by different researchers [ALMZ12, ZLMA12, HMPMS12, IMPMS16].

4.4.3.2. Finding Subset-Minimal Subsets

In some settings one is interested in computing subset-minimal subsets. A well-known example is the extraction and enumeration of minimal unsatisfiable subsets (MUSes) [GMP08, MS10, Nad10, MSL11, BLMS12, NRS13, MSJB13, BK15, NBMS18], or in more general terms, to find minimal explanations for over-constrained systems of constraints. Another example is the extraction and enumeration of minimal correction subsets (MCSes) and their complement: maximal satisfiable subsets (MSSes) [MSHJ⁺13, GLM14a, BDTK14, MPMS15, MIPMS16]. For both MUSes and MCSes, the state of the art algorithms are based on iterative calls to a SAT oracle, with a number of key practical optimizations developed in recent years.

There are many other examples of computing subset-minimal subsets, that exploit CDCL SAT solvers as oracles. Most of these can be cast as instantiations of the Minimal Set over Monotone Predicate (MSMP) problem [MSJB13, MSJM17]. Additional well-known examples include finding minimal models [SI10, SBTLB17], computing the autarky of an unsatisfiable CNF formula [MSIM⁺14, KMS15] and identifying the backbone (i.e. the fixed assignments) of a satisfiable formula [MSJL10, ZWM11, ZWSM11, JLMS15]. Recent examples include [GLM14b, GL15, BGL15, IMMV16], among many others. MSMP has also been exploited in settings where monotonicity does not hold [BIV18].

4.4.3.3. Solving Decision Problems Beyond NP

SAT oracles have achieved significant success in solving decision problems beyond NP. Arguably the most visible have been the stream of improvements made to QBF solvers, by exploiting different forms of abstraction refinement, and also of SAT solvers as oracles [JMS11, JKMSC12, JMS15, RT15, JKMSC16, RS16]. Moreover, the development of an exact pure SAT-based implementation of two-level logic minimization [IPMS15] resulted in an effective alternative to the Quine-McCluskey algorithm [Qui52, Qui55, McC56], the reference two-logic level minimization algorithm for more than five decades.

Abstraction refinement and SAT oracles have been applied to modal logic reasoners [CLLB⁺17, LLBdLM18], but also to propositional abduction [SWJ16, IMMS16] and quantified QBF with soft clauses (QMaxSAT) [IJMS16], among others.

4.4.3.4. Enumerating Sets & Solutions

SAT oracles have enabled observable success in the enumeration of explanations and relaxations of overconstrained systems of constraints. This is the case with the enumeration of minimal unsatisfiable subsets [LPMMS16], the enumeration of minimal correction subsets [PMJaMS18, GIL18], but also the enumeration of prime implicants and implicates [JMSS14, PIMMS15]. A related problem of practical interest is to find the union of all sets, e.g. MUSes or MCSes. One concrete approach is HuMUS [NBE12].

4.4.3.5. Approximate & Exact Solution Counting

Model counting finds a number of important applications. Early work focused on adding dedicated techniques to CDCL SAT solvers to improve exact counting. Concrete examples include connected components and component caching [BJP00, SBK05, GSS09].

More recent work focused on approximate counting, using off-the-shelf CDCL SAT solvers as oracles [EGS12, CMV13a, CMV13b, CMV16, XLE⁺16].

4.4.3.6. Portfolios

The development of portfolios of reasoners [XHHLB08, LBHHX14] has been an active area of research, finding applications for different computational problems. In the case of portfolios for SAT, CDCL SAT solvers represent an integral part of the panoply of solvers that can be used.

4.5. Impact of CDCL SAT Solvers

Over the last two decades, CDCL SAT solving technology has been truly disruptive. This section summarizes the impact of CDCL SAT solvers in terms of well-known practical applications, but also from a theoretical perspective.

4.5.1. Practical Successes

CDCL SAT solvers have been a disruptive technology since their inception, having replaced, or at least complemented, well-established alternative technologies. A number of well-known applications exist but, more importantly, CDCL SAT solvers are in a growing number of cases the engines' engine, i.e. for a number of reasoners in different settings, the most promising approach uses a CDCL SAT solver or mimics the organization of CDCL SAT solvers. This section provides a brief glimpse at the growing range of applications of CDCL SAT solvers.

A few of the best-known hallmark applications of CDCL SAT solvers include:

1. Bounded model checking [BCCZ99, BCC⁺99, Bie09].
2. Unbounded model checking [McM03, Bra11, McM18].
3. Software model checking [CKL04, Kro09].
4. Automated planning [KS99, Rin09].
5. Computer-assisted mathematical proofs [KL14, KL15, HKM16, HK17].

Recent strategic applications of CDCL SAT solvers involve applications in explainable machine learning [IPNaMS18, NIPMS18].

CDCL SAT solvers are also the main engine for a number of reasoners, or have strongly influenced their design. Concrete examples include:

1. MaxSAT (see Section 4.4.3.1)
2. QBF (see Section 4.4.3.3).
3. Modal logics(see Section 4.4.3.3).
4. Satisfiability Modulo Theories (SMT) [BSST09].
5. Answer Set Programming (ASP) [GKKS12].

6. Lazy Clause Generation (LCG) in constraint programming [OSC09].
7. Theorem proving, both with instantiation-based methods [GK03, Kor08], but more recently with superposition theorem provers [KV13, Vor14].

The general setting of model based diagnosis [Rei87], which can be viewed as the problem of explaining overconstrained systems of constraints, finds a wealth of key practical applications [Jun04, BS05, LS08, MSKC14], for which CDCL SAT solving is becoming a key instrument [MSKC14, MSJIM15, AMMS15, LPMMS16].

4.5.2. Theoretical Significance

Recent years have witnessed significant progress in understanding CDCL SAT solvers from a propositional proof complexity perspective. Among the ideas that compose CDCL SAT solvers, from a proof complexity perspective, the key aspects of CDCL SAT solvers are clause learning and search restarts. Whereas the basic backtrack search SAT algorithm (DPLL) is associated with tree-like resolution [BKS04], a proof system weaker than resolution, CDCL SAT solving corresponds instead to a proof system as strong as general resolution [PD09, PD11, AFT09, AFT11]. These recent results build on the clause learning procedure used by CDCL SAT solvers [MS95, MSS96c], namely on the fact that learned clauses are *1-empowering*, and on the use of search restarts [GSC97, GSK98]. A more recent result shows that with a suitable preprocessing technique, restarts can be simulated on a clause learning SAT solver [BS14].

Motivated by the remarkable performance achieved by CDCL SAT solvers, researchers have investigated whether efficient algorithms for stronger proof systems could be devised. This has been the case of extended resolution [Hua10, AKS10], cutting planes [LBP10, EN18, EGCNV18], a recently proposed (dual-rail) MaxSAT-based proof system [IMMS17b, BBI⁺18], but also the recently proposed proof systems related with proof tracing [HKS17, KRPH18].

Although improvements to CDCL SAT solvers continue to be reported, one would expect far more significant impact if practically effective algorithms could be devised for proof systems stronger than general resolution. Although new insights have been reported, it is unclear the success of SAT can be replicated for such stronger proof systems.

4.6. Historical Perspective

The name Conflict-Driven Clause Learning (CDCL) can be traced to L. Ryan's 2004 MSc thesis [Rya04] to designate SAT solvers implementing the DPLL algorithm, but extended with clause learning, search restarts, conflict-aware branching and lazy data structures.

From a theoretical perspective, the most significant contributions of CDCL SAT solvers are clause learning and search restarts.

The clause learning technique used in modern CDCL SAT solvers was proposed in the mid 90s [MS95, MSS96c], with a more extensive treatment in the late 90s [MSS99]. Later work also proposed and used related ideas [BJS97, Zha97].

Nevertheless, all modern CDCL SAT solvers implement the clause learning mechanism first detailed in [MS95, MSS96c], namely by exploiting decision levels and UIPs when learning clauses. The relationship between implication graphs, decision levels, and non-chronological backtracking, without clause learning, was actually described in earlier work [MSS93, MSS94].⁹ In more general settings, learning from conflicts can be traced to the 70s [SS77]. Ideas on backtracking non-chronologically can be traced to about the same time [Gas77, Gas79]. A few additional relevant later works exist [Dec90, Gin93, Pro93]. Nevertheless, and to our best knowledge, in no other area has learning had as significant an impact as in SAT.

The idea of exploiting dominators in conflict analysis is tightly related with the concept of unique sensitization points (USPs) in ATPG [FS83, SA89, MSS94]; both concepts exploit dominators, from which additional inferences can be made. USPs are (static or dynamic) dominators of a circuit with respect to the fault location. UIPs are also dominators, but related with implication sequences. From the inception of UIPs [MS95, MSS96a, MSS96b, MSS96c, MSS99], the name aimed to acknowledge the relationship with USPs.

Search restarts were first proposed in the context of SAT in 1998 [GSC97, GSK98], with the specific purpose of improving the behavior of DPLL-like SAT solvers on satisfiable instances. The practical benefits of exploiting search restarts with clause learning were first documented in 2000 [BMS00], and are now used in any CDCL SAT solver [MMZ⁺01, ES03, Bie08b, AS09]. More recent work offered a theoretical justification for the use of clause learning and search restarts [PD09, PD11, AFT11], showing that it corresponds to a proof system as powerful as general resolution. The concept of *empowering implicate* computed by clause learning SAT solvers [MS95, MSS96c] is crucial for these results.

From a practical perspective, the most significant contributions of CDCL SAT solvers include not only clause learning and search restarts, but also conflict-driven branching and lazy data structures. All these contributions are crucial for the performance of modern CDCL SAT solvers.

The idea of conflict-driven branching can be traced at least to the late 90s [MS99], where the branching heuristics DLIS and DLCS, by dynamically exploiting literal counts, would take into account the literals in already learned clauses. Both DLIS and DLCS were implemented in the GRASP SAT solver since the mid 90s [MS95, MSS96c]. The importance of conflict-driven branching was demonstrated in the Chaff SAT solver [MMZ⁺01], with the VSIDS branching heuristic, which became the de facto reference branching heuristic ever since. Indeed, and even though alternatives have been proposed over the years, VSIDS-like branching is still used in most modern CDCL SAT solvers.

The origins of exploiting lazy data structures in SAT solvers can be traced to 2000 [ZS00], concretely the idea of maintaining head and tail literals in clauses. The much more efficient alternative of watched literals was proposed in the Chaff SAT solver [MMZ⁺01]. With minor modifications, watched literals are still used in modern CDCL SAT solvers.

⁹The name *implication graph* was borrowed from [Lar89, Lar92], where it was used with a different meaning, and which is closely related to the directed graphs associated with 2CNF formulas used in the work Aspvall, Plass and Tarjan [APT79].

In recent times, CDCL is used to mean SAT solvers that implement clause learning, search restarts, conflict-driven branching and lazy data structures, but also a number of commonly used additional techniques. These include phase saving [PD07] but also literal blocks distance [AS09].

4.7. To Probe Further

The chapter provides a comprehensive list of references on the implementation but also one the practical uses of CDCL SAT solvers. Many CDCL SAT solvers have been implemented over the years. The following list summarizes those that are currently publicly available and that have impacted the organization of CDCL SAT solvers.

1. GRASP [MSS96c, MSS99]¹⁰ was the first SAT solver to implement conflict analysis as used in modern CDCL SAT solvers, including clause learning with UIP identification and non-chronological backtracking. Search restarts and its interaction with clause learning were introduced in GRASP at a later stage [BMS00, LBMS01]. GRASP has been open source since its inception.
2. Chaff [MMZ⁺01, ZMMM01]¹¹ introduced efficient lazy data structures, the VSIDS branching heuristic, and also proposed stopping conflict analysis at the first UIP. Chaff is open source, although some of its ideas have been patented [MMM08]¹².
3. MiniSat [ES03]¹³ introduced a simple architecture for the organization of CDCL SAT solvers, one that is still used by most modern CDCL SAT solvers. MiniSat also introduced learned clause minimization in a 2005 release, but this was documented only at a later stage [SB09]. MiniSat has been open source since its inception.
4. Since the mid 00s, A. Biere has developed a number of state of the art SAT solvers, including Lingeling, PicoSAT, PrecoSAT and Limmat¹⁴. These solvers are open source, and some are overviewed in [Bie08b, Bie13, BF15a, Bie16, Bie17].
5. Another CDCL SAT solver is Glucose [AS09]¹⁵, which introduced the LBD metric and proposed ways for using this metric in practice. Glucose builds on the MiniSat code base and it is also open source.
6. A more recent CDCL SAT solver is MapleSAT [LGPC16b, LGPC16a, LOM⁺18].¹⁶ MapleSAT build on the Glucose code base, and it is open source.
7. There are SAT solvers that exploit specific kinds of reasoning. One example is CryptoMiniSat¹⁷ [SNC09]. CryptoMiniSat is open source.

¹⁰GRASP is available from <https://sites.google.com/site/satgrasp/>.

¹¹Chaff is available from <https://www.princeton.edu/~chaff/software.html>.

¹²There have been a number of patents related with CDCL SAT solving and applications. One additional example is related with the work on the BerkMin SAT solver [GN08].

¹³MiniSat is available from github: <https://github.com/niklasso/minisat.git>.

¹⁴Some of A. Biere's SAT solvers are available from: <http://fmv.jku.at/software/>.

¹⁵Glucose is available from: <http://www.labri.fr/perso/lsimon/glucose/>.

¹⁶MapleSAT is available from <https://sites.google.com/a/gsd.uwaterloo.ca/maplesat/>.

¹⁷Available from <https://github.com/msoos/cryptominisat>.

Furthermore, the SAT competitions¹⁸ are an invaluable source of information about state of the art CDCL SAT solvers [BBJS15, HS15, BHJ16, BHJ17a, BHJ17b, HJS18].

A recent effort towards simplifying the implementation of SAT-based tools is the PySAT project¹⁹ [IMMS18]. PySAT offers a Python API to the low-level operation of SAT solvers, as described in previous sections. In turn, this has enabled PySAT-based tools to achieve state of the art performance in a number of settings [IMMS18]. Besides a Python API for interfacing a number of SAT solvers, PySAT offers a variety of encodings of cardinality constraints, which can be helpful for encoding problems into SAT. Furthermore a number of well-known algorithms for solving different function problems are provided as examples of use. These include MUS and MCS extraction, but also MaxSAT solving. The PySAT project is open source, with an MIT license, and should be straightforward to install on most Linux and MacOS distributions.

4.8. Conclusions & Research Directions

This chapter provides an overview of the implementation and practical uses of CDCL SAT solvers. CDCL SAT solving represents one concrete example of remarkable success at solving NP-complete decision problems in practice, and defies the often assumed worst-case characterizations of the run-time of SAT solvers. Regarding the success of practical SAT solving, one of the most striking aspects of this success is that for many years SAT was perceived as an extremely hard problem; all this changed in the mid 90s and early 00s. Similar successes have been witnessed in other areas, including integer linear programming.

As the chapter hopes to demonstrate, there has been a steady stream of improvements to what represents a CDCL SAT solver. Progress is expected to continue. Performance improvements continue to be attained. The application of machine learning has contributed to optimizing components of CDCL SAT solvers, but has also served to develop very effective portfolios of SAT solvers. New insights at the intersection of SAT solver technology and ML are to be expected. The fast pace at which CDCL SAT oracles are being used for problem solving, in different domains, suggests that one should expect that a much wider range of successes of CDCL SAT solving will be witnessed in the near future.

Acknowledgments. Armin Biere, Alexey Ignatiev, and Antonio Morgado provided comments to earlier drafts of this document. Donald Knuth provided detailed comments to the first edition of this chapter.

References

- [ABH⁺08] G. Audemard, L. Bordeaux, Y. Hamadi, S. Jabbour, and L. Sais. A generalized framework for conflict analysis. In *SAT*, volume 4996 of *Lecture Notes in Computer Science*, pages 21–27. Springer, 2008.

¹⁸Detailed information about the SAT competitions since 2002 is available from <https://www.satcompetition.org/>. This website also contains links to the SAT Races of 2006, 2008 and 2018, and the SAT Challenge of 2012.

¹⁹PySAT is available from <https://pysathq.github.io/>.

- [ABL09] C. Ansótegui, M. L. Bonet, and J. Levy. Solving (weighted) partial MaxSAT through satisfiability testing. In *SAT*, pages 427–440, 2009.
- [ABL13] C. Ansótegui, M. L. Bonet, and J. Levy. SAT-based MaxSAT algorithms. *Artif. Intell.*, 196:77–105, 2013.
- [AFT09] A. Atserias, J. K. Fichte, and M. Thurley. Clause-learning algorithms with many restarts and bounded-width resolution. In *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 114–127. Springer, 2009.
- [AFT11] A. Atserias, J. K. Fichte, and M. Thurley. Clause-learning algorithms with many restarts and bounded-width resolution. *J. Artif. Intell. Res.*, 40:353–373, 2011.
- [AKS10] G. Audemard, G. Katsirelos, and L. Simon. A restriction of extended resolution for clause learning SAT solvers. In *AAAI*. AAAI Press, 2010.
- [ALMZ12] C. Ansótegui, C. M. Li, F. Manyà, and Z. Zhu. A SAT-based approach to MinSAT. In *CCIA*, volume 248 of *Frontiers in Artificial Intelligence and Applications*, pages 185–189. IOS Press, 2012.
- [ALS13] G. Audemard, J.-M. Lagniez, and L. Simon. Improving glucose for incremental SAT solving with assumptions: Application to MUS extraction. In *SAT*, volume 7962 of *Lecture Notes in Computer Science*, pages 309–317. Springer, 2013.
- [AM04] C. Ansótegui and F. Manyà. Mapping problems with finite-domain variables to problems with boolean variables. In *SAT*, pages 1–15, 2004.
- [AMMS15] M. F. Arif, C. Mencía, and J. Marques-Silva. Efficient MUS enumeration of horn formulae with applications to axiom pinpointing. In *SAT*, volume 9340 of *Lecture Notes in Computer Science*, pages 324–342. Springer, 2015.
- [ANO⁺12] I. Abío, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and V. Mayer-Eichberger. A new look at BDDs for pseudo-boolean constraints. *J. Artif. Intell. Res.*, 45:443–480, 2012.
- [ANORC09] R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. Cardinality networks and their applications. In *SAT*, pages 167–180, 2009.
- [ANORC11a] I. Abío, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. BDDs for pseudo-boolean constraints - revisited. In *SAT*, pages 61–75, 2011.
- [ANORC11b] R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. Cardinality networks: a theoretical and empirical study. *Constraints*, 16(2):195–221, 2011.
- [APT79] B. Aspvall, M. F. Plass, and R. E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Inf. Process. Lett.*, 8(3):121–123, 1979.
- [AS09] G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI*, pages 399–404, 2009.
- [AS12] G. Audemard and L. Simon. Refining restarts strategies for SAT

- and UNSAT. In M. Milano, editor, *CP*, pages 118–126, 2012.
- [Bat68] K. E. Batchner. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, volume 32 of *AFIPS Conference Proceedings*, pages 307–314. Thomson Book Company, Washington D.C., 1968.
- [BB03] O. Bailleux and Y. Boufkhad. Efficient CNF encoding of boolean cardinality constraints. In *CP*, pages 108–122, 2003.
- [BBI⁺18] M. L. Bonet, S. Buss, A. Ignatiev, J. Marques-Silva, and A. Morgado. MaxSAT resolution with the dual rail encoding. In *AAAI*, pages 6565–6572. AAAI Press, 2018.
- [BBJS15] A. Balint, A. Belov, M. Järvisalo, and C. Sinz. Overview and analysis of the SAT challenge 2012 solver competition. *Artif. Intell.*, 223:120–155, 2015.
- [BBR09] O. Bailleux, Y. Boufkhad, and O. Roussel. New encodings of pseudo-boolean constraints into CNF. In *SAT*, pages 181–194, 2009.
- [BCC⁺99] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *DAC*, pages 317–320. ACM Press, 1999.
- [BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *TACAS*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.
- [BDTK14] F. Bacchus, J. Davies, M. Tsimpoukelli, and G. Katsirelos. Relaxation search: A simple way of managing optional clauses. In *AAAI*, pages 835–841. AAAI Press, 2014.
- [BF15a] A. Biere and A. Fröhlich. Evaluating CDCL restart schemes. In *Sixth Pragmatics of SAT workshop*, 2015.
- [BF15b] A. Biere and A. Fröhlich. Evaluating CDCL variable scoring schemes. In *SAT*, pages 405–422, 2015.
- [BGL15] P. Besnard, É. Grégoire, and J.-M. Lagniez. On computing maximal subsets of clauses that must be satisfiable with possibly mutually-contradictory assumptive contexts. In *AAAI*, pages 3710–3716. AAAI Press, 2015.
- [BHJ16] T. Balyo, M. J. H. Heule, and M. Järvisalo. Proceedings of SAT competition 2016. 2016.
- [BHJ17a] T. Balyo, M. J. H. Heule, and M. Järvisalo. SAT competition 2016: Recent developments. In *AAAI*, pages 5061–5063. AAAI Press, 2017.
- [BHJ17b] T. Balyo, M. H. Heule, and M. Järvisalo. Proceedings of SAT competition 2017: Solver and benchmark descriptions. 2017.
- [BHvMW09] A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.
- [Bie08a] A. Biere. Adaptive restart strategies for conflict driven SAT solvers. In *SAT*, volume 4996 of *Lecture Notes in Computer Science*, pages 28–33. Springer, 2008.
- [Bie08b] A. Biere. PicoSAT essentials. *JSAT*, 4(2-4):75–97, 2008.

- [Bie09] A. Biere. Bounded model checking. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 457–481. IOS Press, 2009.
- [Bie13] A. Biere. Lingeling, plingeling and treengeling entering the SAT competition 2013. *Proceedings of SAT competition*, 51, 2013.
- [Bie16] A. Biere. Splatz, lingeling, plingeling, treengeling, yalsat entering the sat competition 2016. *Proc. of SAT Competition*, pages 44–45, 2016.
- [Bie17] A. Biere. Cadical, lingeling, plingeling, treengeling and yalsat entering the sat competition 2017. *SAT COMPETITION 2017*, page 14, 2017.
- [BIV18] R. Berryhill, A. Ivrii, and A. G. Veneris. Finding all minimal safe inductive sets. In *SAT*, pages 346–362, 2018.
- [BJP00] R. J. Bayardo Jr. and J. D. Pehoushek. Counting models using connected components. In *AAAI*, pages 157–162, 2000.
- [BJRS18] A. Boudane, S. Jabbour, B. Raddaoui, and L. Sais. Efficient sat-based encodings of conditional cardinality constraints. In *LPAR*, volume 57 of *EPiC Series in Computing*, pages 181–195. EasyChair, 2018.
- [BJS97] R. J. Bayardo Jr. and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *AAAI*, pages 203–208, 1997.
- [BK15] F. Bacchus and G. Katsirelos. Using minimal correction sets to more efficiently compute minimal unsatisfiable sets. In *CAV (2)*, volume 9207 of *Lecture Notes in Computer Science*, pages 70–86. Springer, 2015.
- [BKN⁺09] C. Bessière, G. Katsirelos, N. Narodytska, C.-G. Quimper, and T. Walsh. Decompositions of all different, global cardinality and related constraints. In *IJCAI*, pages 419–424, 2009.
- [BKS04] P. Beame, H. A. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res.*, 22:319–351, 2004.
- [BL99] H. K. Büning and T. Lettmann. *Propositional logic: deduction and algorithms*. Cambridge University Press, 1999.
- [BLMS12] A. Belov, I. Lynce, and J. Marques-Silva. Towards efficient MUS extraction. *AI Commun.*, 25(2):97–116, 2012.
- [BMS00] L. Baptista and J. Marques-Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In *CP*, volume 1894 of *Lecture Notes in Computer Science*, pages 489–494. Springer, 2000.
- [BMS12] L. Bordeaux and J. Marques-Silva. Knowledge compilation with empowerment. In *SOFSEM*, volume 7147 of *Lecture Notes in Computer Science*, pages 612–624. Springer, 2012.
- [Bra11] A. R. Bradley. SAT-based model checking without unrolling. In *VMCAI*, pages 70–87, 2011.
- [BS05] J. Bailey and P. J. Stuckey. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In *PADL*, pages 174–186, 2005.

- [BS14] P. Beame and A. Sabharwal. Non-restarting SAT solvers with simple preprocessing can efficiently simulate resolution. In *AAAI*, pages 2608–2615. AAAI Press, 2014.
- [BSST09] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.
- [CESS08] K. Claessen, N. Een, M. Sheeran, and N. Sorensson. Sat-solving in practice. In *WODES*, pages 61–67. IEEE, 2008.
- [CFHHJ⁺17] L. Cruz-Filipe, M. J. H. Heule, W. A. Hunt Jr., M. Kaufmann, and P. Schneider-Kamp. Efficient certified RAT verification. In *CADE*, volume 10395 of *Lecture Notes in Computer Science*, pages 220–236. Springer, 2017.
- [CFMSSK17] L. Cruz-Filipe, J. Marques-Silva, and P. Schneider-Kamp. Efficient certified resolution proof checking. In *TACAS (1)*, volume 10205 of *Lecture Notes in Computer Science*, pages 118–135, 2017.
- [CGLR96] J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *KR*, pages 148–159. Morgan Kaufmann, 1996.
- [CKL04] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.
- [CLLB⁺17] T. Caridroit, J.-M. Lagniez, D. Le Berre, T. de Lima, and V. Montmirail. A SAT-based approach for solving the modal logic S5-satisfiability problem. In *AAAI*, pages 3864–3870, 2017.
- [CMV13a] S. Chakraborty, K. S. Meel, and M. Y. Vardi. A scalable and nearly uniform generator of SAT witnesses. In *CAV*, pages 608–623, 2013.
- [CMV13b] S. Chakraborty, K. S. Meel, and M. Y. Vardi. A scalable approximate model counter. In *CP*, pages 200–216, 2013.
- [CMV16] S. Chakraborty, K. S. Meel, and M. Y. Vardi. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT calls. In *IJCAI*, pages 3569–3576, 2016.
- [Coo71] S. A. Cook. The complexity of theorem-proving procedures. In *STOC*, pages 151–158. ACM, 1971.
- [CS05] M. Cadoli and A. Schaerf. : Compiling problem specifications into SAT. *Artif. Intell.*, 162(1-2):89–120, 2005.
- [CZI10] M. Codish and M. Zazon-Ivry. Pairwise cardinality networks. In *LPAR (Dakar)*, volume 6355 of *Lecture Notes in Computer Science*, pages 154–172. Springer, 2010.
- [DB11] J. Davies and F. Bacchus. Solving MAXSAT by solving a sequence of simpler SAT instances. In *CP*, pages 225–239, 2011.
- [Dec90] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artif. Intell.*, 41(3):273–312, 1990.
- [DFMS10] A. Darbari, B. Fischer, and J. Marques-Silva. Industrial-strength certified SAT solving through verified SAT proof checking. In *IC-TAC*, volume 6255 of *Lecture Notes in Computer Science*, pages

- 260–274. Springer, 2010.
- [DJN09] R. Drechsler, T. A. Junntila, and I. Niemelä. Non-clausal SAT and ATPG. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 655–693. IOS Press, 2009.
- [DLL62] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.
- [DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.
- [EGCNV18] J. Elffers, J. Giráldez-Cru, J. Nordström, and M. Vinyals. Using combinatorial benchmarks to probe the reasoning power of pseudo-boolean solvers. In *SAT*, pages 75–93, 2018.
- [EGS12] S. Ermon, C. P. Gomes, and B. Selman. Uniform solution sampling using a constraint solver as an oracle. In *UAI*, pages 255–264, 2012.
- [EN15] A. Erez and A. Nadel. Finding bounded path in graph using SMT for automatic clock routing. In *CAV*, pages 20–36, 2015.
- [EN18] J. Elffers and J. Nordström. Divide and conquer: Towards faster pseudo-boolean solving. In *IJCAI*, pages 1291–1299, 2018.
- [ES03] N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT*, pages 502–518, 2003.
- [ES06] N. Eén and N. Sörensson. Translating pseudo-boolean constraints into SAT. *JSAT*, 2(1-4):1–26, 2006.
- [FBS19] K. Fazekas, A. Biere, and C. Scholl. Incremental inprocessing in SAT solving. In M. Janota and I. Lynce, editors, *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, volume 11628 of *Lecture Notes in Computer Science*, pages 136–154. Springer, 2019.
- [FM06] Z. Fu and S. Malik. On solving the partial MAX-SAT problem. In *SAT*, volume 4121 of *Lecture Notes in Computer Science*, pages 252–265. Springer, 2006.
- [FP01] A. M. Frisch and T. J. Peugniez. Solving non-boolean satisfiability problems with stochastic local search. In *IJCAI*, pages 282–290. Morgan Kaufmann, 2001.
- [FS83] H. Fujiwara and T. Shimono. On the acceleration of test generation algorithms. *IEEE Trans. Computers*, 32(12):1137–1144, 1983.
- [Gas77] J. Gaschnig. A general backtrack algorithm that eliminates most redundant tests. In *International Joint Conference on Artificial Intelligence*, page 457, 1977.
- [Gas79] J. Gaschnig. *Performance measurement and analysis of certain search algorithms*. PhD thesis, Carnegie-Mellon University, 1979.
- [Gen02] I. P. Gent. Arc consistency in SAT. In *ECAI*, pages 121–125. IOS Press, 2002.
- [Gen13] I. P. Gent. Optimal implementation of watched literals and more general techniques. *J. Artif. Intell. Res.*, 48:231–251, 2013.
- [GeSSMS99] L. Guerra e Silva, L. M. Silveira, and J. Marques-Silva. Algorithms for solving boolean satisfiability in combinational circuits.

- In *DATE*, pages 526–530. IEEE Computer Society / ACM, 1999.
- [GIL18] É. Grégoire, Y. Izza, and J.-M. Lagniez. Boosting mcses enumeration. In *IJCAI*, pages 1309–1315, 2018.
- [Gin93] M. L. Ginsberg. Dynamic backtracking. *J. Artif. Intell. Res.*, 1:25–46, 1993.
- [GK03] H. Ganzinger and K. Korovin. New directions in instantiation-based theorem proving. In *LICS*, pages 55–64, 2003.
- [GKKS12] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.
- [GKSS08] C. P. Gomes, H. A. Kautz, A. Sabharwal, and B. Selman. Satisfiability solvers. In *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*, pages 89–134. Elsevier, 2008.
- [GL15] É. Grégoire and J.-M. Lagniez. On anti-subsumptive knowledge enforcement. In *LPAR*, volume 9450 of *Lecture Notes in Computer Science*, pages 48–62. Springer, 2015.
- [GLM14a] É. Grégoire, J.-M. Lagniez, and B. Mazure. An experimentally efficient method for (mss, comss) partitioning. In *AAAI*, pages 2666–2673, 2014.
- [GLM14b] É. Grégoire, J.-M. Lagniez, and B. Mazure. Multiple contraction through Partial-Max-SAT. In *ICTAI*, pages 321–327. IEEE Computer Society, 2014.
- [GMP08] É. Grégoire, B. Mazure, and C. Piette. On approaches to explaining infeasibility of sets of boolean clauses. In *ICTAI*, pages 74–83, 2008.
- [GN02] E. I. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *DATE*, pages 142–149. IEEE Computer Society, 2002.
- [GN03] E. I. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *DATE*, pages 10886–10891. IEEE Computer Society, 2003.
- [GN04] I. P. Gent and P. Nightingale. A new encoding of alldifferent into SAT. In *International Workshop on Modelling and Reformulating Constraint Satisfaction*, pages 95–110, 2004.
- [GN08] E. Goldberg and Y. Novikov. Method and system for solving satisfiability problems, April 8 2008. US Patent 7,356,519.
- [GS09] C. P. Gomes and A. Sabharwal. Exploiting runtime variation in complete solvers. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 271–288. IOS Press, 2009.
- [GSC97] C. P. Gomes, B. Selman, and N. Crato. Heavy-tailed distributions in combinatorial search. In *CP*, volume 1330 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 1997.
- [GSCK00] C. P. Gomes, B. Selman, N. Crato, and H. A. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *J. Autom. Reasoning*, 24(1/2):67–100, 2000.
- [GSK98] C. P. Gomes, B. Selman, and H. A. Kautz. Boosting combinatorial search through randomization. In *AAAI/IAAI*, pages 431–437.

- AAAI Press / The MIT Press, 1998.
- [GSS09] C. P. Gomes, A. Sabharwal, and B. Selman. Model counting. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 633–654. IOS Press, 2009.
- [Heu18] M. J. H. Heule. Schur number five. In *AAAI*, pages 6598–6606, 2018.
- [HHJKW17] M. J. H. Heule, W. A. Hunt Jr., M. Kaufmann, and N. Wetzler. Efficient, verified checking of propositional proofs. In *ITP*, volume 10499 of *Lecture Notes in Computer Science*, pages 269–284. Springer, 2017.
- [HHJW13] M. J. H. Heule, W. A. Hunt Jr., and N. Wetzler. Trimming while checking clausal proofs. In *FMCAD*, pages 181–188. IEEE, 2013.
- [HHJW14] M. J. H. Heule, W. A. Hunt Jr., and N. Wetzler. Bridging the gap between easy generation and efficient verification of unsatisfiability proofs. *Softw. Test., Verif. Reliab.*, 24(8):593–607, 2014.
- [HHW13] M. J. H. Heule, W. A. Hunt Jr., and N. Wetzler. Verifying refutations with extended resolution. In *CADE*, volume 7898 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 2013.
- [HJS18] M. J. H. Heule, M. Järvisalo, and M. Suda. Proceedings of SAT competition 2018. 2018.
- [HK17] M. J. H. Heule and O. Kullmann. The science of brute force. *Commun. ACM*, 60(8):70–79, 2017.
- [HKM16] M. J. H. Heule, O. Kullmann, and V. W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In N. Creignou and D. Le Berre, editors, *SAT*, volume 9710 of *Lecture Notes in Computer Science*, pages 228–245. Springer, 2016.
- [HKSB17] M. J. H. Heule, B. Kiesl, M. Seidl, and A. Biere. PRuning through satisfaction. In *Haiifa Verification Conference*, volume 10629 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2017.
- [HMPMS12] F. Heras, A. Morgado, J. Planes, and J. Marques-Silva. Iterative SAT solving for minimum satisfiability. In *ICTAI*, pages 922–927. IEEE Computer Society, 2012.
- [HS15] M. J. H. Heule and T. Schaub. What’s hot in the SAT and ASP competitions. In *AAAI*, pages 4322–4323, 2015.
- [Hua07] J. Huang. The effect of restarts on the efficiency of clause learning. In *IJCAI*, pages 2318–2323, 2007.
- [Hua10] J. Huang. Extended clause learning. *Artif. Intell.*, 174(15):1277–1284, 2010.
- [IJMS16] A. Ignatiev, M. Janota, and J. Marques-Silva. Quantified maximum satisfiability. *Constraints*, 21(2):277–302, 2016.
- [IMM⁺14] A. Ignatiev, A. Morgado, V. M. Manquinho, I. Lynce, and J. Marques-Silva. Progression in maximum satisfiability. In *ECAI*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 453–458. IOS Press, 2014.
- [IMMS16] A. Ignatiev, A. Morgado, and J. Marques-Silva. Propositional abduction with implicit hitting sets. In *ECAI*, volume 285 of *Frontiers in Artificial Intelligence and Applications*, pages 1327–1335.

- IOS Press, 2016.
- [IMMS17a] A. Ignatiev, A. Morgado, and J. Marques-Silva. Cardinality encodings for graph optimization problems. In *IJCAI*, pages 652–658, 2017.
- [IMMS17b] A. Ignatiev, A. Morgado, and J. Marques-Silva. On tackling the limits of resolution in SAT solving. In *SAT*, volume 10491 of *Lecture Notes in Computer Science*, pages 164–183. Springer, 2017.
- [IMMS18] A. Ignatiev, A. Morgado, and J. Marques-Silva. PySAT: A python toolkit for prototyping with SAT oracles. In *SAT*, volume 10929 of *Lecture Notes in Computer Science*, pages 428–437. Springer, 2018.
- [IMMV16] A. Ivrii, S. Malik, K. S. Meel, and M. Y. Vardi. On computing minimal independent support and its applications to sampling and counting. *Constraints*, 21(1):41–58, 2016.
- [IMPMS16] A. Ignatiev, A. Morgado, J. Planes, and J. Marques-Silva. Maximal falsifiability. *AI Commun.*, 29(2):351–370, 2016.
- [IPMS15] A. Ignatiev, A. Previti, and J. Marques-Silva. SAT-based formula simplification. In *SAT*, volume 9340 of *Lecture Notes in Computer Science*, pages 287–298. Springer, 2015.
- [IPNaMS18] A. Ignatiev, F. Pereira, N. Narodytska, and J. ao Marques-Silva. A SAT-based approach to learn explainable decision sets. In *IJCAR*, volume 10900 of *Lecture Notes in Computer Science*, pages 627–645. Springer, 2018.
- [Jac00] D. Jackson. Automating first-order relational logic. In *FSE*, pages 130–139, 2000.
- [JaMS16] M. Janota and J. ao Marques-Silva. On the query complexity of selecting minimal sets for monotone predicates. *Artif. Intell.*, 233:73–83, 2016.
- [JBH10] M. Järvisalo, A. Biere, and M. J. H. Heule. Blocked clause elimination. In *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 129–144. Springer, 2010.
- [JHB12] M. Järvisalo, M. J. H. Heule, and A. Biere. Inprocessing rules. In *IJCAR*, volume 7364 of *Lecture Notes in Computer Science*, pages 355–370. Springer, 2012.
- [JJ09] M. Järvisalo and T. A. Junttila. Limitations of restricted branching in clause learning. *Constraints*, 14(3):325–356, 2009.
- [JKMSC12] M. Janota, W. Klieber, J. Marques-Silva, and E. M. Clarke. Solving QBF with counterexample guided refinement. In *SAT*, volume 7317 of *Lecture Notes in Computer Science*, pages 114–128. Springer, 2012.
- [JKMSC16] M. Janota, W. Klieber, J. Marques-Silva, and E. M. Clarke. Solving QBF with counterexample guided refinement. *Artif. Intell.*, 234:1–25, 2016.
- [JLMS15] M. Janota, I. Lynce, and J. Marques-Silva. Algorithms for computing backbones of propositional formulae. *AI Commun.*, 28(2):161–177, 2015.
- [JLSS14] S. Jabbour, J. Lonlac, L. Sais, and Y. Salhi. Extending modern SAT solvers for models enumeration. In *IRI*, pages 803–810. IEEE

- Computer Society, 2014.
- [JMS11] M. Janota and J. Marques-Silva. Abstraction-based algorithm for 2QBF. In *SAT*, volume 6695 of *Lecture Notes in Computer Science*, pages 230–244. Springer, 2011.
- [JMS15] M. Janota and J. Marques-Silva. Solving QBF by clause selection. In *IJCAI*, pages 325–331. AAAI Press, 2015.
- [JMSS14] S. Jabbour, J. Marques-Silva, L. Sais, and Y. Salhi. Enumerating prime implicants of propositional formulae in conjunctive normal form. In *JELIA*, volume 8761 of *Lecture Notes in Computer Science*, pages 152–165. Springer, 2014.
- [Jun04] U. Junker. QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems. In *AAAI*, pages 167–172, 2004.
- [Kar18] M. Karpinski. Encoding cardinality constraints using standard encoding of generalized selection networks preserves arc-consistency. *Theor. Comput. Sci.*, 707:77–81, 2018.
- [KGP01] A. Kuehlmann, M. K. Ganai, and V. Paruthi. Circuit-based boolean reasoning. pages 232–237, 2001.
- [KKM94] R. Kohli, R. Krishnamurti, and P. Mirchandani. The minimum satisfiability problem. *SIAM J. Discrete Math.*, 7(2):275–283, 1994.
- [KL14] B. Konev and A. Lisitsa. A SAT attack on the Erdős discrepancy conjecture. In *SAT*, volume 8561 of *Lecture Notes in Computer Science*, pages 219–226. Springer, 2014.
- [KL15] B. Konev and A. Lisitsa. Computer-aided proof of Erdős discrepancy properties. *Artif. Intell.*, 224:103–118, 2015.
- [KMS15] O. Kullmann and J. Marques-Silva. Computing maximal autarkies with few and simple oracle queries. In *SAT*, volume 9340 of *Lecture Notes in Computer Science*, pages 138–155. Springer, 2015.
- [Knu15] D. E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Addison-Wesley Professional, 1st edition, 2015.
- [Kor08] K. Korovin. iprover - an instantiation-based theorem prover for first-order logic (system description). In *IJCAIR*, pages 292–298, 2008.
- [Kro09] D. Kroening. Software verification. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 505–532. IOS Press, 2009.
- [KRPH18] B. Kiesl, A. Rebola-Pardo, and M. J. H. Heule. Extended resolution simulates DRAT. In *IJCAR*, volume 10900 of *Lecture Notes in Computer Science*, pages 516–531. Springer, 2018.
- [KS92] H. A. Kautz and B. Selman. Planning as satisfiability. In *ECAI*, pages 359–363, 1992.
- [KS96] H. A. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In *AAAI*, pages 1194–1201, 1996.
- [KS99] H. A. Kautz and B. Selman. Unifying SAT-based and graph-based planning. In *IJCAI*, pages 318–325. Morgan Kaufmann, 1999.

- [KSTW05] P. Kilby, J. K. Slaney, S. Thiébaux, and T. Walsh. Backbones and backdoors in satisfiability. In *AAAI*, pages 1368–1373. AAAI Press / The MIT Press, 2005.
- [KV13] L. Kovács and A. Voronkov. First-order theorem proving and vampire. In *CAV*, pages 1–35, 2013.
- [Lar89] T. Larrabee. Efficient generation of test patterns using boolean difference. In *ITC*, pages 795–802, 1989.
- [Lar92] T. Larrabee. Test pattern generation using boolean satisfiability. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 11(1):4–15, 1992.
- [LB13] J.-M. Lagniez and A. Biere. Factoring out assumptions to speed up MUS extraction. In *SAT*, volume 7962 of *Lecture Notes in Computer Science*, pages 276–292. Springer, 2013.
- [LBHGX14] K. Leyton-Brown, H. H. Hoos, F. Hutter, and L. Xu. Understanding the empirical hardness of NP-complete problems. *Commun. ACM*, 57(5):98–107, 2014.
- [LBMS01] I. Lynce, L. Baptista, and J. Marques-Silva. Towards provably complete stochastic search algorithms for satisfiability. In *EPIA*, volume 2258 of *Lecture Notes in Computer Science*, pages 363–370. Springer, 2001.
- [LBP10] D. Le Berre and A. Parrain. The Sat4j library, release 2.2. *JSAT*, 7(2-3):59–6, 2010.
- [Lev73] L. A. Levin. Universal sequential search problems. *Problemy Peredachi Informatsii*, 9(3):115–116, 1973.
- [LGPC16a] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki. Exponential recency weighted average branching heuristic for SAT solvers. In *AAAI*, pages 3434–3440, 2016.
- [LGPC16b] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki. Learning rate based branching heuristic for SAT solvers. In *SAT*, pages 123–140, 2016.
- [LLBdLM18] J.-M. Lagniez, D. Le Berre, T. de Lima, and V. Montmi-rail. An assumption-based approach for solving the minimal S5-satisfiability problem. In *IJCAR*, pages 1–18, 2018.
- [LLX⁺17] M. Luo, C.-M. Li, F. Xiao, F. Manyà, and Z. Lü. An effective learnt clause minimization approach for CDCL SAT solvers. In *IJCAI*, pages 703–711, 2017.
- [LM09] C. M. Li and F. Manyà. MaxSAT, hard and soft constraints. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 613–631. IOS Press, 2009.
- [LMQZ10] C. M. Li, F. Manyà, Z. Quan, and Z. Zhu. Exact MinSAT solving. In *SAT*, volume 6175 of *Lecture Notes in Computer Science*, pages 363–368. Springer, 2010.
- [LMS02] I. Lynce and J. Marques-Silva. Building state-of-the-art SAT solvers. In *ECAI*, pages 166–170. IOS Press, 2002.
- [LMS03] I. Lynce and J. Marques-Silva. An overview of backtrack search satisfiability algorithms. *Ann. Math. Artif. Intell.*, 37(3):307–326, 2003.

- [LOM⁺18] J. H. Liang, C. Oh, M. Mathew, C. Thomas, C. Li, and V. Ganesh. Machine learning-based restart policy for CDCL SAT solvers. In *SAT*, pages 94–110, 2018.
- [LPMMS16] M. H. Liffiton, A. Previti, A. Malik, and J. Marques-Silva. Fast, flexible MUS enumeration. *Constraints*, 21(2):223–250, 2016.
- [LS08] M. H. Liffiton and K. A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning*, 40(1):1–33, 2008.
- [LSZ93] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of las vegas algorithms. *Inf. Process. Lett.*, 47(4):173–180, 1993.
- [MB19] S. Möhle and A. Biere. Backing backtracking. In *SAT*, pages 250–266, 2019.
- [McC56] E. J. McCluskey. Minimization of Boolean functions. *Bell system technical Journal*, 35(6):1417–1444, 1956.
- [McM03] K. L. McMillan. Interpolation and sat-based model checking. In *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.
- [McM18] K. L. McMillan. Interpolation and model checking. In *Handbook of Model Checking*, pages 421–446. Springer, 2018.
- [MDMS14] A. Morgado, C. Dodaro, and J. Marques-Silva. Core-guided MaxSAT with soft cardinality constraints. In *CP*, volume 8656 of *Lecture Notes in Computer Science*, pages 564–573. Springer, 2014.
- [MHB12] N. Manthey, M. J. H. Heule, and A. Biere. Automated reencoding of boolean formulas. In *Haiifa Verification Conference*, volume 7857 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2012.
- [MHL⁺13] A. Morgado, F. Heras, M. H. Liffiton, J. Planes, and J. Marques-Silva. Iterative and core-guided MaxSA solving: A survey and assessment. *Constraints*, 18(4):478–534, 2013.
- [MIPMS16] C. Mencia, A. Ignatiev, A. Previti, and J. Marques-Silva. MCS extraction with sublinear oracle queries. In *SAT*, volume 9710 of *Lecture Notes in Computer Science*, pages 342–360. Springer, 2016.
- [Mit05] D. G. Mitchell. A SAT solver primer. *Bulletin of the EATCS*, 85:112–132, 2005.
- [MJML14] R. Martins, S. Joshi, V. M. Manquinho, and I. Lynce. Incremental cardinality constraints for maxsat. In *CP*, volume 8656 of *Lecture Notes in Computer Science*, pages 531–548. Springer, 2014.
- [MMM08] M. Moskewicz, C. Madigan, and S. Malik. Method and system for efficient implementation of boolean satisfiability. Available from <https://patents.google.com/patent/US7418369B2/en>, August 26 2008. US Patent 7,418,369.
- [MMSP09] V. M. Manquinho, J. Marques-Silva, and J. Planes. Algorithms for weighted boolean optimization. In *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 495–508. Springer, 2009.
- [MMZ⁺01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, pages 530–535.

- ACM, 2001.
- [MPMS15] C. Mencía, A. Previti, and J. Marques-Silva. Literal-based MCS extraction. In *IJCAI*, pages 1973–1979. AAAI Press, 2015.
- [MPS14] N. Manthey, T. Philipp, and P. Steinke. A more compact translation of pseudo-boolean constraints into CNF such that generalized arc consistency is maintained. In *KI*, volume 8736 of *Lecture Notes in Computer Science*, pages 123–134. Springer, 2014.
- [MS95] J. P. Marques-Silva. *Search Algorithms for Satisfiability Problems in Combinational Switching Circuits*. PhD thesis, University of Michigan, May 1995.
- [MS98] J. Marques-Silva. An overview of backtrack search satisfiability algorithms. In *ISAIM*, 1998.
- [MS99] J. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *EPIA*, volume 1695 of *Lecture Notes in Computer Science*, pages 62–74. Springer, 1999.
- [MS08] J. Marques-Silva. Practical applications of boolean satisfiability. In *WODES*, pages 74–80. IEEE, 2008.
- [MS10] J. Marques-Silva. Minimal unsatisfiability: Models, algorithms and applications (invited paper). In *ISMVL*, pages 9–14. IEEE Computer Society, 2010.
- [MSGeS03] J. Marques-Silva and L. Guerra e Silva. Solving satisfiability in combinational circuits. *IEEE Design & Test of Computers*, 20(4):16–21, 2003.
- [MSHJ⁺13] J. Marques-Silva, F. Heras, M. Janota, A. Previti, and A. Belov. On computing minimal correction subsets. In *IJCAI*, pages 615–622. IJCAI/AAAI, 2013.
- [MSIM⁺14] J. Marques-Silva, A. Ignatiev, A. Morgado, V. M. Manquinho, and I. Lynce. Efficient autarkies. In *ECAI*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 603–608. IOS Press, 2014.
- [MSJB13] J. Marques-Silva, M. Janota, and A. Belov. Minimal sets over monotone predicates in boolean formulae. In *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 592–607. Springer, 2013.
- [MSJIM15] J. Marques-Silva, M. Janota, A. Ignatiev, and A. Morgado. Efficient model based diagnosis with maximum satisfiability. In *IJCAI*, pages 1966–1972. AAAI Press, 2015.
- [MSJL10] J. Marques-Silva, M. Janota, and I. Lynce. On computing backbones of propositional theories. In *ECAI*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 15–20. IOS Press, 2010.
- [MSJM17] J. Marques-Silva, M. Janota, and C. Mencía. Minimal sets on propositional formulae. problems and reductions. *Artif. Intell.*, 252:22–50, 2017.
- [MSKC14] A. Metodi, R. Stern, M. Kalech, and M. Codish. A novel sat-based approach to model based diagnosis. *J. Artif. Intell. Res.*, 51:377–411, 2014.
- [MSL11] J. Marques-Silva and I. Lynce. On improving MUS extraction al-

- gorithms. In *SAT*, volume 6695 of *Lecture Notes in Computer Science*, pages 159–173. Springer, 2011.
- [MSLM09] J. Marques-Silva, I. Lynce, and S. Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 131–153. IOS Press, 2009.
- [MSM18] J. Marques-Silva and S. Malik. Propositional SAT solving. In *Handbook of Model Checking*, pages 247–275. Springer, 2018.
- [MSP07] J. Marques-Silva and J. Planes. On using unsatisfiability for solving maximum satisfiability. *CoRR*, abs/0712.1097, 2007.
- [MSS⁺91] P. C. McGeer, A. Saldanha, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Timing analysis and delay-fault test generation using path-recursive functions. In *ICCAD*, pages 180–183. IEEE Computer Society, 1991.
- [MSS93] J. Marques-Silva and K. A. Sakallah. Space pruning heuristics for path sensitization in test pattern generation. Technical Report CSE-TR-178-93, University of Michigan, 1993.
- [MSS94] J. Marques-Silva and K. A. Sakallah. Dynamic search-space pruning techniques in path sensitization. In *DAC*, pages 705–711. ACM Press, 1994.
- [MSS96a] J. Marques-Silva and K. A. Sakallah. Conflict analysis in search algorithms for propositional satisfiability. Technical Report RT-04-96, INESC, May 1996.
- [MSS96b] J. Marques-Silva and K. A. Sakallah. Grasp – a new search algorithm for satisfiability. Technical Report CSE-TR-292-96, University of Michigan, April 1996.
- [MSS96c] J. Marques-Silva and K. A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.
- [MSS99] J. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.
- [MSS00] J. Marques-Silva and K. A. Sakallah. Boolean satisfiability in electronic design automation. In *DAC*, pages 675–680. ACM, 2000.
- [MZ09] S. Malik and L. Zhang. Boolean satisfiability from theoretical hardness to practical success. *Commun. ACM*, 52(8):76–82, 2009.
- [MZK⁺99] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. Determining computational complexity from characteristic ‘phase transitions’. *Nature*, 400(6740):133, 1999.
- [Nad10] A. Nadel. Boosting minimal unsatisfiable core extraction. In *FM-CAD*, pages 221–229, 2010.
- [NBE12] A. Nöhler, A. Biere, and A. Egyed. Managing SAT inconsistencies with HUMUS. In *VaMoS*, pages 83–91. ACM, 2012.
- [NBMS18] N. Narodytska, N. Bjørner, M.-C. Marinescu, and M. Sagiv. Core-guided minimal correction set and core enumeration. In *IJCAI*, pages 1353–1361, 2018.
- [NIPMS18] N. Narodytska, A. Ignatiev, F. Pereira, and J. Marques-Silva. Learning optimal decision trees with SAT. In *IJCAI*, pages 1362–

- 1368, 2018.
- [NR12] A. Nadel and V. Ryvchin. Efficient SAT solving under assumptions. In *SAT*, pages 242–255, 2012.
- [NR18] A. Nadel and V. Ryvchin. Chronological backtracking. In *SAT*, pages 111–121, 2018.
- [NRS13] A. Nadel, V. Ryvchin, and O. Strichman. Efficient MUS extraction with resolution. In *FMCAD*, pages 197–200, 2013.
- [NRS14] A. Nadel, V. Ryvchin, and O. Strichman. Ultimately incremental SAT. In *SAT*, pages 206–218, 2014.
- [Oh15] C. Oh. Between SAT and UNSAT: the fundamental difference in CDCL SAT. In *SAT*, pages 307–323, 2015.
- [OLH⁺13] T. Ogawa, Y. Liu, R. Hasegawa, M. Koshimura, and H. Fujita. Modulo based CNF encoding of cardinality constraints and its application to MaxSAT solvers. In *ICTAI*, pages 9–17. IEEE Computer Society, 2013.
- [OSC09] O. Ohrimenko, P. J. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.
- [PD07] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In *SAT*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer, 2007.
- [PD09] K. Pipatsrisawat and A. Darwiche. On the power of clause-learning SAT solvers with restarts. In *CP*, volume 5732 of *Lecture Notes in Computer Science*, pages 654–668. Springer, 2009.
- [PD11] K. Pipatsrisawat and A. Darwiche. On the power of clause-learning SAT solvers as resolution engines. *Artif. Intell.*, 175(2):512–525, 2011.
- [PIMMS15] A. Previti, A. Ignatiev, A. Morgado, and J. Marques-Silva. Prime compilation of non-clausal formulae. In *IJCAI*, pages 1980–1988. AAAI Press, 2015.
- [PMJaMS18] A. Previti, C. Mencía, M. Järvisalo, and J. ao Marques-Silva. Premise set caching for enumerating minimal correction subsets. In *AAAI*, pages 6633–6640. AAAI Press, 2018.
- [Pre07] S. D. Prestwich. Variable dependency in local search: Prevention is better than cure. In *SAT*, pages 107–120, 2007.
- [Pre09] S. D. Prestwich. CNF encodings. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 75–97. IOS Press, 2009.
- [Pre15] A. Previti. *On the Duality Relating Sets*. PhD thesis, University College Dublin, 2015.
- [Pro93] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.
- [Qui52] W. V. Quine. The problem of simplifying truth functions. *American mathematical monthly*, pages 521–531, 1952.
- [Qui55] W. V. Quine. A way to simplify truth functions. *American mathematical monthly*, pages 627–631, 1955.
- [Rei87] R. Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, 1987.

- [Rin09] J. Rintanen. Planning and SAT. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 483–504. IOS Press, 2009.
- [RM09] O. Roussel and V. M. Manquinho. Pseudo-boolean and cardinality constraints. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 695–733. IOS Press, 2009.
- [Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.
- [RS16] M. N. Rabe and S. A. Seshia. Incremental determinization. In *SAT*, pages 375–392, 2016.
- [RT15] M. N. Rabe and L. Tentrup. CAQE: A certifying QBF solver. In *FMCAD*, pages 136–143, 2015.
- [Rya04] L. Ryan. Efficient algorithms for clause-learning SAT solvers. Master’s thesis, Simon Fraser University, February 2004.
- [SA89] M. H. Schulz and E. Auth. Improved deterministic test pattern generation with applications to redundancy identification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 8(7):811–816, 1989.
- [Sak09] K. A. Sakallah. Symmetry and satisfiability. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 289–338. IOS Press, 2009.
- [SB09] N. Sörensson and A. Biere. Minimizing learned clauses. In *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 237–243. Springer, 2009.
- [SBJ16] P. Saikko, J. Berg, and M. Järvisalo. LMHS: A SAT-IP hybrid MaxSAT solver. In *SAT*, pages 539–546, 2016.
- [SBK05] T. Sang, P. Beame, and H. A. Kautz. Heuristics for fast exact model counting. In *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 226–240. Springer, 2005.
- [SBSV96] P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 15(9):1167–1176, 1996.
- [SBTLB17] T. Soh, M. Banbara, N. Tamura, and D. Le Berre. Solving multiobjective discrete optimization problems with propositional minimal model generation. In *CP*, volume 10416 of *Lecture Notes in Computer Science*, pages 596–614. Springer, 2017.
- [SI09] C. Sinz and M. Iser. Problem-sensitive restart heuristics for the DPLL procedure. In *SAT*, pages 356–362, 2009.
- [SI10] T. Soh and K. Inoue. Identifying necessary reactions in metabolic pathways by minimal model generation. In *ECAI*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 277–282. IOS Press, 2010.
- [Sin05] C. Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In *CP*, pages 827–831, 2005.
- [SLM92] B. Selman, H. J. Levesque, and D. G. Mitchell. A new method for solving hard satisfiability problems. In *AAAI*, pages 440–446,

- 1992.
- [SNC09] M. Soos, K. Nohl, and C. Castelluccia. Extending SAT solvers to cryptographic problems. In *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009.
 - [SP04] S. Subbarayan and D. K. Pradhan. NiVER: Non increasing variable elimination resolution for preprocessing SAT instances. In *SAT*, 2004.
 - [SS77] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):135–196, 1977.
 - [SSS12] A. Sabharwal, H. Samulowitz, and M. Sellmann. Learning backclauses in SAT. In *SAT*, pages 498–499, 2012.
 - [Stu13] P. J. Stuckey. There are no CNF problems. In *SAT*, pages 19–21, 2013.
 - [SWJ16] P. Saikko, J. P. Wallner, and M. Järvisalo. Implicit hitting set algorithms for reasoning beyond NP. In *KR*, pages 104–113, 2016.
 - [SZGN17] X. Si, X. Zhang, R. Grigore, and M. Naik. Maximum satisfiability in software analysis: Applications and techniques. In *CAV*, pages 68–94, 2017.
 - [SZO⁺18] A. A. Semenov, O. Zaikin, I. V. Otpuschennikov, S. Kochemazov, and A. Ignatiev. On cryptographic attacks using backdoors for SAT. In *AAAI*, pages 6641–6648. AAAI Press, 2018.
 - [TBW04] C. Thiffault, F. Bacchus, and T. Walsh. Solving non-clausal formulas with DPLL search. In *CP*, volume 3258 of *Lecture Notes in Computer Science*, pages 663–678. Springer, 2004.
 - [TJ07] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *TACAS*, pages 632–647, 2007.
 - [Van08] A. Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *ISAIM*, 2008.
 - [Van12] A. Van Gelder. Producing and verifying extremely large propositional refutations - have your cake and eat it too. *Ann. Math. Artif. Intell.*, 65(4):329–372, 2012.
 - [VG02] A. Van Gelder. Extracting (easily) checkable proofs from a satisfiability solver that employs both preorder and postorder resolution. In *ISAIM*, 2002.
 - [VG09] A. Van Gelder. Improved conflict-clause minimization leads to improved propositional proof traces. In *SAT*, pages 141–146, 2009.
 - [VG11] A. Van Gelder. Generalized conflict-clause strengthening for satisfiability solvers. In *SAT*, pages 329–342, 2011.
 - [Vor14] A. Voronkov. AVATAR: the architecture for first-order theorem provers. In *CAV*, pages 696–710, 2014.
 - [VWM15] Y. Vizel, G. Weissenbacher, and S. Malik. Boolean satisfiability solvers and their applications in model checking. *Proceedings of the IEEE*, 103(11):2021–2035, 2015.
 - [WA09] T. Weber and H. Amjad. Efficiently checking propositional refutations in HOL theorem provers. *J. Applied Logic*, 7(1):26–40, 2009.
 - [Wal00] T. Walsh. SAT v CSP. In *CP*, volume 1894 of *Lecture Notes in*

- Computer Science*, pages 441–456. Springer, 2000.
- [War98] J. P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Inf. Process. Lett.*, 68(2):63–69, 1998.
- [Web06] T. Weber. Efficiently checking propositional resolution proofs in Isabelle/HOL. In *IWIL*, pages 44–62, 2006.
- [WGS03] R. Williams, C. P. Gomes, and B. Selman. Backdoors to typical case complexity. In *IJCAI*, pages 1173–1178. Morgan Kaufmann, 2003.
- [WHHJ13] N. Wetzler, M. J. H. Heule, and W. A. Hunt Jr. Mechanical verification of SAT refutations with extended resolution. In *ITP*, volume 7998 of *Lecture Notes in Computer Science*, pages 229–244. Springer, 2013.
- [WHHJ14] N. Wetzler, M. J. H. Heule, and W. A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *SAT*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, 2014.
- [WvM98] J. P. Warners and H. van Maaren. A two-phase algorithm for solving a class of hard satisfiability problems. *Oper. Res. Lett.*, 23(3-5):81–88, 1998.
- [XHHLB08] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Satzilla: Portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res.*, 32:565–606, 2008.
- [XLE⁺16] Y. Xue, Z. Li, S. Ermon, C. P. Gomes, and B. Selman. Solving marginal MAP problems with NP oracles and parity constraints. In *NIPS*, pages 1127–1135, 2016.
- [Zha97] H. Zhang. SATO: an efficient propositional prover. In *CADE*, pages 272–275, 1997.
- [ZLMA12] Z. Zhu, C. M. Li, F. Manyà, and J. Argelich. A new encoding from MinSAT into MaxSAT. In *CP*, volume 7514 of *Lecture Notes in Computer Science*, pages 455–463. Springer, 2012.
- [ZM88] R. Zabih and D. A. McAllester. A rearrangement search strategy for determining propositional satisfiability. In *AAAI*, pages 155–160, 1988.
- [ZM02] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2002.
- [ZM03] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *DATE*, pages 10880–10885. IEEE Computer Society, 2003.
- [ZMMM01] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285. IEEE Computer Society, 2001.
- [ZS00] H. Zhang and M. E. Stickel. Implementing the Davis-Putnam method. *J. Autom. Reasoning*, 24(1/2):277–296, 2000.
- [ZWM11] C. S. Zhu, G. Weissenbacher, and S. Malik. Post-silicon fault local-

isation using maximum satisfiability and backbones. In *FMCAD*, pages 63–66. FMCAD Inc., 2011.

- [ZWSM11] C. S. Zhu, G. Weissenbacher, D. Sethi, and S. Malik. SAT-based techniques for determining backbones for post-silicon fault localisation. In *HLDVT*, pages 84–91. IEEE Computer Society, 2011.

Chapter 5

Look-Ahead Based SAT Solvers

Marijn J.H. Heule and Hans van Maaren

5.1. Introduction

Imagine that you are for the first time in New York City (NYC). A cab just dropped you off at a crossing and now you want to see the most beautiful part of town. You consider two potential strategies to get going: A *conflict-driven* strategy and a *look-ahead* strategy.

The conflict-driven strategy consists of the following heuristics: On each crossing you look in all directions and walk towards the crossing which appears most beautiful at first sight. If at a certain crossing all new directions definitely lead to dead end situations, you write the location in a booklet to avoid the place in the future and you evaluate which was the nearest crossing where you likely chose the wrong direction. You go back to that crossing and continue with a new preferred direction.

The look-ahead strategy spends much more time to select the next crossing. First all adjacent crossings are visited. At each of them all directions are observed before returning to the current crossing. Now the next crossing is selected based on all observed directions. A schematic view of both strategies is shown in Figure 5.1.

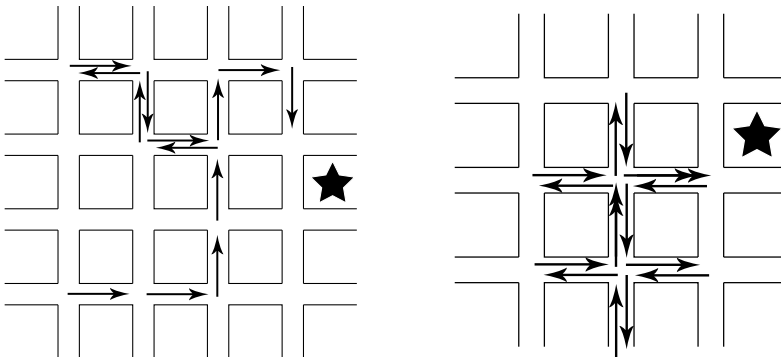


Figure 5.1. A NYC walk, conflict-driven (left) and look-ahead (right). ★ denotes the target.

As the reader has probably expected by now, the conflict-driven and look-ahead SAT solving architectures have several analogies with the conflict-driven and look-ahead NYC sightseeing strategies. Although the look-ahead NYC walk appears very costly, in practice it solves some problems quite efficiently. This chapter walks through the look-ahead architecture for SAT solvers.

5.1.1. Domain of application

Before going into the details of the look-ahead architecture, we first consider a wide range of problems and ask the question: Which architecture is the most appropriate to adequately solve a particular instance?

Traditionally, look-ahead SAT solvers are strong on random k -SAT formulae and especially on the unsatisfiable instances. Herwig [Her06] suggests that in practice, look-ahead SAT solvers are strong on benchmarks with either a low density (ratio clauses to variables) or a small diameter (longest shortest path in for instance the resolution graph¹). Figure 5.2 illustrates this. Using the structured (crafted and industrial) benchmarks from the SAT 2005 competition and SATlib, the relative performance is measured for the solvers MiniSAT (conflict-driven) and march (look-ahead). For a given combination of density and diameter of the resolution graph, the strongest solver is plotted.

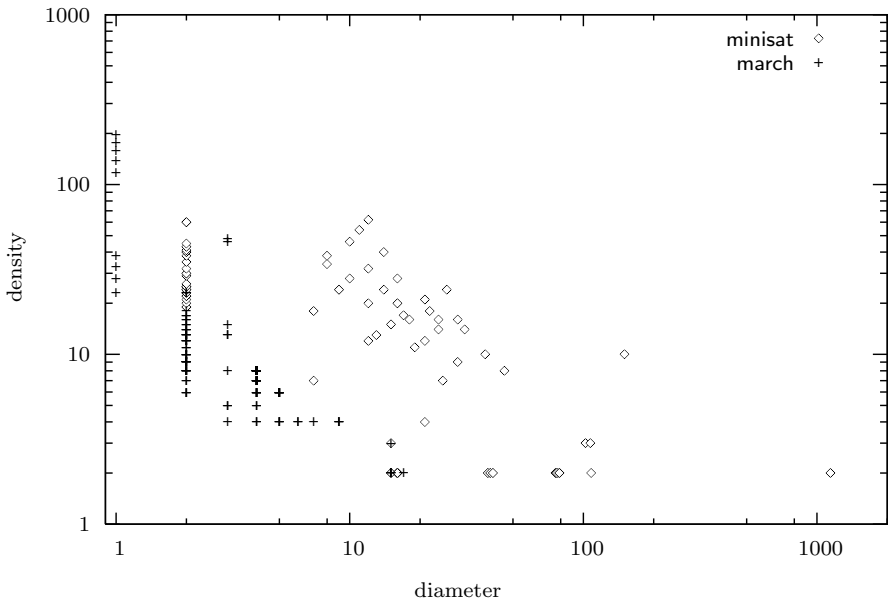


Figure 5.2. Strongest architecture on structured benchmarks split by density and diameter. Architectures represented by MiniSAT (conflict-driven) and march (look-ahead).

¹The resolution graph is a clause based graph. Its vertices are the clauses and clauses are connected if they have exactly one clashing literal.

Notice that Figure 5.2 compares MiniSAT to `march` and therefore it should not be interpreted blindly as a comparison between any pair of conflict-driven and look-ahead SAT solvers in general. The solver `march` is the only look-ahead SAT solver that is optimized for large and structured benchmarks. Selecting a different look-ahead SAT solver would change the picture in favor of MiniSAT.

However, it seems that in the current state of development, both architectures have their own range of applications. The almost sharp separation shown in Figure 5.2 can be explained as follows: Density expresses the cost of unit propagation. The higher the density, the more clauses need to be reduced for each assigned variable. Therefore, look-ahead becomes very expensive for formulae with high densities, while using lazy data-structures (like MiniSAT), the cost of unit propagation does not increase heavily on these formulae.

The diameter expresses the global connectivity of clauses. If just a few variables cover all the literals of a set of clauses, such a set is referred to as a *local cluster*. The larger the diameter, the more multiple local clusters occur within the formula. Local clusters reduce the effectiveness of reasoning by look-ahead SAT solvers: Assigning decision variables to a truth value will modify the formula only locally. Therefore, expensive reasoning is only expected to learn facts within the last modified cluster. On the other hand, conflict-driven solvers benefit from local clusters: Conflict clauses will likely arise from local conflicts, yielding small clauses consisting of some covering variables. Also, they may be reused frequently.

5.2. General and Historical Overview

5.2.1. The Look-Ahead Architecture

The look-ahead architecture is based on the *DPLL* framework [DLL62]: It is a complete solving method which selects in each step a decision variable x_{decision} and recursively calls DPLL for the reduced formula where x_{decision} is assigned to false (denoted by $\mathcal{F}[x_{\text{decision}} = 0]$) and another where x_{decision} is assigned to true (denoted by $\mathcal{F}[x_{\text{decision}} = 1]$).

A formula \mathcal{F} is reduced by *unit propagation*: Given a formula \mathcal{F} , an unassigned variable x and a Boolean value \mathbf{B} , first x is assigned to \mathbf{B} . If this assignment φ results in a *unit clause* (clause of size 1), then φ is expanded by assigning the remaining literal of that clause to true. This is repeated until no unit clauses are left in φ applied to \mathcal{F} (denoted by $\varphi \circ \mathcal{F}$) - see Algorithm 5.1. The reduced formula consists of all clauses that are not satisfied. So, $\mathcal{F}[x = \mathbf{B}] := \varphi \circ \mathcal{F}$.

Algorithm 5.1 UNITPROPAGATION (formula \mathcal{F} , variable x , $\mathbf{B} \in \{0, 1\}$)

```

1:  $\varphi := \{x \leftarrow \mathbf{B}\}$ 
2: while empty clause  $\perp \notin \varphi \circ \mathcal{F}$  and unit clause  $y \in \varphi \circ \mathcal{F}$  do
3:    $\varphi := \varphi \cup \{y \leftarrow 1\}$ 
4: end while
5: return  $\varphi \circ \mathcal{F}$ 

```

The recursion has two kinds of leaf nodes: Either all clauses have been satisfied (denoted by $\mathcal{F} = \emptyset$), meaning that a satisfying assignment has been found, or \mathcal{F} contains an *empty clause* \perp (a clause of which all literals have been falsified), meaning a dead end. In the latter case the algorithm backtracks.

The decision variable is selected by the LOOKAHEAD procedure. Besides selecting x_{decision} it also attempts to reduce the formula in each step by assigning *forced* variables and to further constrain it by adding clauses. Algorithm 5.2 shows the top level structure. Notice that the LOOKAHEAD procedure returns both a simplified formula and x_{decision} .

In addition, the presented algorithm uses direction heuristics to select which of the reduced formulae $\mathcal{F}[x_{\text{decision}} = 0]$ or $\mathcal{F}[x_{\text{decision}} = 1]$ should be visited first - see Section 5.3.2. Effective direction heuristics improve the performance on satisfiable instances². Although direction heuristics can be applied to all DPLL based solvers, look-ahead SAT solvers are particularly the ones that use them.

Algorithm 5.2 DPLL (formula \mathcal{F})

```

1: if  $\mathcal{F} = \emptyset$  then
2:   return satisfiable
3: end if
4:  $\langle \mathcal{F}; x_{\text{decision}} \rangle := \text{LOOKAHEAD}(\mathcal{F})$ 
5: if empty clause  $\perp \in \mathcal{F}$  then
6:   return unsatisfiable
7: else if no  $x_{\text{decision}}$  is selected then
8:   return DPLL( $\mathcal{F}$ )
9: end if
10:  $\mathbf{B} := \text{DIRECTIONHEURISTIC}(x_{\text{decision}}, \mathcal{F})$ 
11: if DPLL( $\mathcal{F}[x_{\text{decision}} = \mathbf{B}]$ ) = satisfiable then
12:   return satisfiable
13: end if
14: return DPLL( $\mathcal{F}[x_{\text{decision}} = \neg \mathbf{B}]$ )

```

The LOOKAHEAD procedure, as the name suggests, performs *look-aheads*. A look-ahead on x starts by assigning x to true followed by unit propagation. The importance of x is measured *and* possible reductions of the formula are detected. After this analysis, it backtracks, ending the look-ahead. The rationale of a look-ahead operation is that evaluating the effect of actually assigning variables to truth values and performing unit propagation is more adequate compared to taking a cheap guess using some statistical data on \mathcal{F} .

This brings us to the two main features of the LOOKAHEAD procedure. The first is the *decision heuristic* which measures the importance of a variable. This heuristic consists of a *difference* or *distance heuristic* (in short DIFF) and a heuristic that combines two DIFF values (in short MIXDIFF).

A DIFF heuristic measures the reduction of the formulae caused by a look-ahead. The larger the reduction, the higher the heuristic value. This reduction can be measured with all kinds of statistics. Effective statistics are the reduction of free variables and the number of newly created (reduced, but not satisfied) clauses. The final judgment of a variable is obtained by combining $\text{DIFF}(\mathcal{F}, \mathcal{F}[x = 0])$ and $\text{DIFF}(\mathcal{F}, \mathcal{F}[x = 1])$ using a MIXDIFF heuristic. The product of these numbers is generally considered to be an effective heuristic. It aims to create a balanced search-tree. Notice that this is not a goal in its own: The trivial 2^n tree is perfectly balanced, but huge. See Chapter 8 for more details and theory.

²Direction heuristics applied in a conflict-driven setting may heavily influence performance on unsatisfiable formulae.

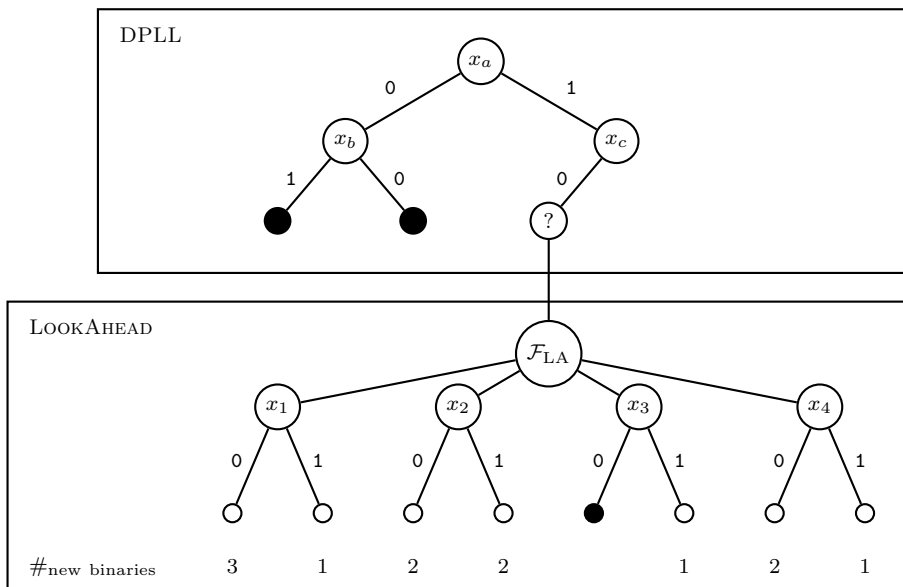


Figure 5.3. A graphical representation of the look-ahead architecture. Above, the DPLL super-structure (a binary tree) is shown. In each node of the DPLL-tree, the LOOKAHEAD procedure is called to select the decision variable and to compute implied variables by additional reasoning. Black nodes refer to leaf nodes and variables shown in the vertices refer to the decision variables and look-ahead variables, respectively.

The second main feature of the look-ahead architecture is the detection of *failed literals*: If a look-ahead on x results in a conflict, then x is forced to be assigned to false. Detection of failed literals reduces the formula because of these “free” assignments. The LOOKAHEAD procedure terminates in case both x_i and $\neg x_i$ are detected as failed literals.

Figure 5.3 shows a graphical representation of the look-ahead architecture. On the top level, the DPLL framework is used. The selection of the decision variable, reduction of the formula, and addition of learned clauses are performed by the LOOKAHEAD procedure. Black nodes refer to a dead end situation, either an unsatisfiable leaf node (DPLL) or a failed literal (LOOKAHEAD procedure).

Example 5.2.1. Consider the following example formula:

$$\mathcal{F}_{LA} = (\neg x_1 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_4) \wedge (x_1 \vee \neg x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee x_4)$$

Since the largest clauses in \mathcal{F}_{LA} have size three, only new binary clauses can be created. For instance, during the look-ahead on $\neg x_1$, three new binary clauses are created (all clauses in which literal x_1 occurs). The look-ahead on x_1 will force x_3 to be assigned to true by unit propagation. This will reduce the last clause to a binary clause, while all other clauses become satisfied. Similarly, we can compute the number of new binary clauses (denoted by #_{new binaries}) for all look-aheads - see Figure 5.3.

Notice that the look-ahead on $\neg x_3$ results in a conflict. So $\neg x_3$ is a *failed literal* and forces x_3 to be assigned to true. Due to this forced assignment the formula changes. To improve the accuracy of the look-ahead heuristics (in this case the reduction measurement), the look-aheads should be performed again. However, by assigning forced variables, more failed literals might be detected. So, for accuracy, first iteratively perform the look-aheads until no new failed literals are detected.

Finally, the selection of the decision variable is based on the reduction measurements of both the look-ahead on $\neg x_i$ and x_i . Generally, the product is used to combine the numbers. In this example, x_2 would be selected as decision variable, because the product of the reduction measured while performing look-ahead on $\neg x_2$ and x_2 is the highest (i.e. 4). Section 8.6 discusses the preference for the product in more detail.

Several enhancements of the look-ahead architecture have been developed. To reduce the cost of the LOOKAHEAD procedure, look-aheads could be restricted to a subset of the free variables. The subset (denoted by \mathcal{P}) is selected by the PRESELECT procedure - see Section 5.3.3. However, if $|\mathcal{P}|$ is too small, this could decrease overall performance, since less failed literals will be detected and possibly a less effective decision variable is selected.

Various forms of additional look-ahead reasoning can be applied to \mathcal{F} to either reduce its size by assigning forced literals or to further constrain it by adding learned clauses. Four kinds of additional reasoning are discussed in Section 5.4.

Algorithm 5.3 LOOKAHEAD (\mathcal{F})

```

1:  $\mathcal{P} := \text{PRESELECT}(\mathcal{F})$ 
2: repeat
3:   for all variables  $x_i \in \mathcal{P}$  do
4:      $\mathcal{F} := \text{LOOKAHEADREASONING}(\mathcal{F}, x)$ 
5:     if empty clause  $\perp \in \mathcal{F}[x = 0]$  and empty clause  $\perp \in \mathcal{F}[x = 1]$  then
6:       return  $\langle \perp; * \rangle$ 
7:     else if empty clause  $\perp \in \mathcal{F}[x = 0]$  then
8:        $\mathcal{F} := \mathcal{F}[x = 1]$ 
9:     else if empty clause  $\perp \in \mathcal{F}[x = 1]$  then
10:       $\mathcal{F} := \mathcal{F}[x = 0]$ 
11:    else
12:       $H(x_i) = \text{DECISIONHEURISTIC}(\mathcal{F}, \mathcal{F}[x = 0], \mathcal{F}[x = 1])$ 
13:    end if
14:  end for
15: until nothing (important) has been learned
16: return  $\langle \mathcal{F}; x_i \text{ with greatest } H(x_i) \rangle$ 

```

Algorithm 5.3 shows the LOOKAHEAD procedure with these enhancements. Mostly the procedure will return a simplified formula \mathcal{F} and a decision variable x_{decision} . Except in two cases this procedure returns no decision variable: First, if the procedure detects that the formula is unsatisfiable by failed literals. Second, if all (pre-selected) variables are assigned. In the latter case, we either detected a satisfying assignment or we need to restart the LOOKAHEAD procedure with a new set of pre-selected variables.

Detection of failed literals or other look-ahead reasoning that change \mathcal{F} , can influence the DIFF values. Also the propagation of a forced literal could result

in the failure of the look-head on other literals. Therefore, to improve both the accuracy of the decision heuristic and to increase the number of detected failed literals, the LOOKAHEAD procedure can be iterated until no (important) facts are learned (in the last iteration).

The unit propagation part within the LOOKAHEAD procedure is relatively the most costly aspect of this architecture. Heuristics aside, performance could be improved by reduction of these costs. Section 5.5 discusses three optimization techniques.

5.2.2. History of Look-Ahead SAT Solvers

The Böhm solver [BS96] could be considered as the first look-ahead SAT solver. In each node it selects the variable that occurs most frequently in the shortest active (not satisfied) clauses. Although no look-aheads are performed, it uses *eager data-structures* which are common in look-ahead SAT solvers: Two dimensional linked lists are used to cheaply compute the occurrences in active clauses. This data-structure is also used in the OKsolver. The Böhm solver won the first SAT competition in 1991/1992 [BK92].

The first SAT solvers with a LOOKAHEAD procedure were developed in the mid 1990s: *csat* by Dubois et al. [DABC93] and *posit* by Freeman [Fre95, Fre96]. They already contained aspects of the important heuristics used in the “modern” look-ahead SAT solvers:

- Decision variables are selected by a DIFF heuristic that measures the reduction of free variables. The MIXDIFF heuristic used in *posit*, is still used in most look-ahead SAT solvers. Let $L := \text{DIFF}(\mathcal{F}, \mathcal{F}[x = 0])$ and $R := \text{DIFF}(\mathcal{F}, \mathcal{F}[x = 1])$ then MIXDIFF is computed by $1024 \cdot LR + L + R$. The product is selected to create a balanced search-tree. The addition is for tie-breaking purposes. The factor 1024, although seemingly arbitrary, gives priority to the product (balancedness of the emerging search-tree).
- A direction heuristic is used to improve the performance on satisfiable instances: Using a simple heuristic, it estimates the relative costs of solving $\mathcal{F}[x = 0]$ and $\mathcal{F}[x = 1]$. The smallest one is preferred as first branch.
- Variables in *posit* are pre-selected for the LOOKAHEAD procedure using three principles: First, important variables (ones with a high estimated MIXDIFF) are selected to get an effective decision variable. Second, literals that occur frequently negated in binary clauses are selected (so possibly not their complement) as candidate failed literals. Third, many variables are selected near the root of the tree, because here, accurate heuristics and failed literals have more impact on the size of the tree.

Results on hard random 3-SAT formulae were boosted by *satz*. This solver was developed by Li and Anbulagan in 1997. All heuristics were optimized for these benchmarks:

- The DIFF heuristic measures the newly created binary clauses in a weighted manner based on the literals in these binary clauses. This heuristic is called *weighted binarities heuristic* and is discussed in Section 5.3.1.2. The same MIXDIFF heuristic is used as in *posit*.

- No direction heuristics are used. $\mathcal{F}[x_{\text{decision}} = 1]$ is always preferred.
- Variables are pre-selected using the prop_z heuristic [LA97a] - see Section 5.3.3.1. On random 3-SAT formulae, it pre-selects most free variables near the root of the search-tree. As the tree deepens, the number of selected variables decreases.

In 1999, Li added a `DOUBLELOOK` procedure to `satz` and further reduced the size of the search-tree to solve random formulae [Li99]. This procedure - see Section 5.4.3 - attempts to detect more failed literals by also performing some look-aheads on a second level of propagation: If during the look-ahead on x more than a certain number of new binary clauses are created, then additional look-aheads are performed on the reduced formula $\mathcal{F}[x = 1]$ to check whether it is unsatisfiable. The parameter (called Δ_{trigger}) which expresses the number of new binary clauses to trigger the `DOUBLELOOK` procedure is fixed to 65 based on experiments on hard random 3-SAT formulae. Performance clearly improves using this setting, while on many structured benchmarks it results in a slowdown: The procedure is executed too frequently.

To exploit the presence of so-called equivalence clauses in many benchmarks, Li [Li00] created a special version of `satz`, called `eqsatz`, which uses equivalence reasoning. During the look-ahead phase, it searches binary equivalences $x_i \leftrightarrow x_j$ in the reduced formulae $\mathcal{F}[x = 0]$ and $\mathcal{F}[x = 1]$. Using five inference rules, it reasons about the detected equivalences. Due to this equivalence reasoning, `eqsatz` was the first SAT solver to solve the hard `parity32` benchmarks [CKS95] without performing Gaussian elimination on the detected equivalence clauses.

Starting from 1998, Oliver Kullmann developed the `OKsolver`, motivation by the idea to create a “pure” look-ahead SAT solver [Kul02]. The heuristics are kept clean. They are not stuffed with magic constants:

- The `DIFF` heuristic in `OKsolver` measures the newly created clauses in a weighted manner - based on the size on the new clauses. For each of the used weights a separate experiment has been performed. This *clause reduction heuristic* is discussed in Section 5.3.1.1. The `MIXDIFF` uses the product $\text{DIFF}(\mathcal{F}, \mathcal{F}[x = 0]) \cdot \text{DIFF}(\mathcal{F}, \mathcal{F}[x = 1])$ which is based on the τ -function [KL99].
- The direction heuristics prefer $\mathcal{F}[x_{\text{decision}} = 0]$ or $\mathcal{F}[x_{\text{decision}} = 1]$ which is probabilistically the most satisfiable. Details are shown in Section 5.3.2.
- No pre-selection heuristics are used.

Besides the clean heuristics, `OKsolver` also adds some reasoning to the look-ahead architecture:

- **Local Learning**³: If the look-ahead on x assigns y to true, then we learn $x \rightarrow y$. This can be stored as binary clause $\neg x \vee y$. These learned binary clauses are valid under the partial assignment. This means that they should be removed while backtracking.
- **Autarky Reasoning**: If during the look-ahead on x no new clauses are created then an autarky is detected. This means that $\mathcal{F}[x = 1]$ is satisfi-

³Local learning is an optimal feature in the `OKsolver` which was turned off in the version submitted to SAT 2002.

ability equivalent with \mathcal{F} . Therefore, x can be freely assigned to true. If only one new clause is created, then an 1-autarky is detected. In this case, two local learned binary clauses can be added. Details are presented in Section 5.4.2.

- **Backjumping:** While backtracking, the OKsolver maintains a list of the variables that were responsible for detected conflicts. If a decision variable does not occur in the list, the second branch can be skipped. This technique has the same effect as the ones used in conflict-driven solvers.

To maximize the number of learned facts, the LOOKAHEAD procedure is iterated until in the last iteration no new forced literals have been detected and no new binary clauses have been added. During the first SAT competition⁴ (2002), OKsolver won both divisions of random k -SAT benchmarks.

The first version of `march`, developed by Heule et al. [HDvZvM04] in 2002 was inspired by `satz` and applied most of its features. The most significant difference was a pre-processing technique which translated any formula to 3-SAT. The updated version of 2003, `march_eq`, was modified to become a look-ahead SAT solver for general purposes. New techniques were added to increase performance on structured instances, while still being competitive on random formulae. Three techniques have been added to improve efficiency. Due to some overhead these techniques are cost neutral on random formulae, but generate significant speed-ups on many structured instances:

- The binary and non-binary clauses of the formula are stored in separate data-structures. This reduces the required memory (especially on structured benchmarks) and facilitates a more efficient unit propagation - see Section 5.5.1.
- Before entering the LOOKAHEAD procedure, all redundant (satisfied) clauses are removed from the active data-structures. This boosts the unit propagation during this procedure - see Section 5.5.2.
- By building implication trees of the binary clauses containing two pre-selected variables, a lot of redundancy in the LOOKAHEAD procedure can be tackled - see Section 5.5.3.

Also equivalence reasoning was added. Instead of searching for equivalence clauses in all reduced formulae, `march_eq` only extracts them from the input formula. They are stored in a separate data-structure to implement a specialized unit propagation procedure. The solver won two divisions of crafted benchmarks during SAT 2004.

The `kcfnfs` solver by Dubois and Dequen, developed in 2004, is in many aspects similar to `satz`. However, it features some important improvements with respect to hard random k -SAT formulae:

- The most effective modification is a minor one. Like `satz`, the DIFF heuristic measures the newly created binary clauses in a weighted manner - based on occurrences of the variables in these clauses. However, instead of adding the weighted literals, they are multiplied per binary clause - see Section 5.3.1.3. This *backbone search heuristic* (BSH) significantly improves

⁴see www.satcompetition.org

the performance on random k -SAT formulae.

- The implementation is optimized for efficiency on random k -SAT formulae.
- Normalization has been developed for *BSH*, such that it can measure newly created clauses of any size. This is required for fast performance on random k -SAT formulae with $k > 3$.

The *kcfnfs* solver performs very strong on random formulae on the SAT competitions. It won one random division during both SAT 2003 and SAT 2004, and even two random divisions during SAT 2005.

In 2005 two adaptive heuristics were added to *march_eq*, resulting in *march_dl*:

- It was observed [HvM06a] that there is some correlation between the number of failed literals and the optimal number of pre-selected free variables. Based on this observation, an adaptive heuristic was added which aims to converge to the optimal value.
- An adaptive algorithm for the *DOUBLELOOK* procedure [HvM07] has been added to *march_eq*. It modifies Δ_{trigger} after each look-ahead. In contrast to earlier implementations of the *DOUBLELOOK* procedure, it now reduces the computational costs on almost all formulae.

5.3. Heuristics

As the previous section showed, the look-ahead architecture enables one to invoke many heuristic features. Research tends to focus on three main categories of heuristics, which we recall here:

- **Difference heuristics:** To measure the difference between the formulae before and after a look-ahead. The quality of the used difference heuristic influences the actual impact of the decision variable.
- **Direction heuristics:** Given a decision variable x_{decision} , one can choose whether to examine first the positive branch $\mathcal{F}[x_{\text{decision}} = 1]$ or the negative branch $\mathcal{F}[x_{\text{decision}} = 0]$. Effective direction heuristics improve the performance on satisfiable formulae.
- **Pre-selection heuristics:** To reduce the computational costs, look-ahead can be restricted to a subset of the variables. As a possible negative consequence, however, a smaller pre-selected set of variables may result in fewer detected failed literals and a less effective decision variable.

5.3.1. Difference heuristics

This section covers the branching heuristics used in look-ahead SAT solvers. A general description of these heuristics is offered in Chapter 8. To measure the difference between a formula and its reduction after a look-ahead (in short *DIFF*), various kinds of statistics can be applied. For example: The reduction of free variables, the reduction of the number of clauses, the reduced size of clauses, or any combination. *Posit* uses the reduced number of free variables as *DIFF* [Fre95]. All look-ahead SAT solvers which participated in the SAT competitions use a *DIFF* based on newly created (reduced, but not satisfied) clauses. The set of new

clauses created during the look-ahead on x is denoted by $\mathcal{F}[x = 1] \setminus \mathcal{F}$. All these heuristics use weights to quantify the relative importance of clauses of a certain size. The importance of a clause of size k is denoted by γ_k and the subformula of \mathcal{F} which contains only the clauses of size k is denoted by \mathcal{F}_k . This section discusses and compares three main DIFF heuristics.

5.3.1.1. Clause reduction heuristic

The DIFF implemented by Kullmann in OKsolver [Kul02] uses only the γ_k weights and is called *clause reduction heuristic*⁵ (*CRH*):

$$CRH(x_i) := \sum_{k \geq 2} \gamma_k \cdot |\mathcal{F}[x_i = 1]_k \setminus \mathcal{F}| \quad \text{with}$$

$$\gamma_2 := 1, \gamma_3 := 0.2, \gamma_4 := 0.05, \gamma_5 := 0.01, \gamma_6 := 0.003$$

$$\gamma_k := 20.4514 \cdot 0.218673^k \text{ for } k \geq 7$$

These constants γ_k for $k = 3, 4, 5, 6$ are the result of performance optimization of the OKsolver on random k -SAT formulae, while the formula for $k \geq 7$ is the result of linear regression [Kul02].

5.3.1.2. Weighted binaries heuristic

Weighted binaries heuristic (*WBH*) is developed by Li and Anbulagan [LA97b] and applied in the solver **satz**. Each variable is weighted (both positive and negative) based on its occurrences in the formula. Each new binary clause $x \vee y$ is weighted using the sum of the weights of their complementary literals. The sum $\#(\neg x) + \#(\neg y)$ expresses the number of clauses on which resolution can be done with $x \vee y$.

Occurrences in a clause of size k is valued 5^{k-3} . This weighting function followed from performance optimization of **satz** over random formulae. Notice that as in the OKsolver (i.e. γ_k in *CRH*) clauses of size $k + 1$ are about $\frac{1}{5}$ of the importance of clauses of size k - although a different heuristic is used on a different solver.

$$w_{WBH}(x_i) := \sum_{k \geq 2} \gamma_k \cdot \#_k(x_i) \quad \text{with } \gamma_k := 5^{3-k}$$

$$WBH(x_i) := \sum_{(x \vee y) \in \{\mathcal{F}[x_i = 1]_2 \setminus \mathcal{F}_2\}} (w_{WBH}(\neg x) + w_{WBH}(\neg y))$$

5.3.1.3. Backbone search heuristic

Backbone search heuristic (*BSH*) was developed by Dubois and Dequen [DD01]. This heuristic is inspired by the concept of the *backbone* of a formula - the set of

⁵This heuristic is also known as *MNC* and is further discussed in Section 8.7.4.1.

variables that has a fixed truth value in all assignments satisfying the maximum number of clauses [MZK⁺99]. Like *WBH*, the variables are weighted based on their occurrences in clauses of various sizes. Clauses of size $k + 1$ are considered only half⁶ as important as clauses of size k .

The most influential difference between *WBH* and *BSH* is that the latter multiplies the weights of the literals in the newly created binary clauses - while the former adds them. For every new clause $x \vee y$, the product $\#(\neg x) \cdot \#(\neg y)$ expresses the number of resolvents that can be created due to the new clause.

$$w_{BSH}(x_i) := \sum_{k \geq 2} \gamma_k \cdot \#_k(x_i) \quad \text{with } \gamma_k := 2^{3-k}$$

$$BSH(x_i) := \sum_{(x \vee y) \in \mathcal{F}[x_i=1]_2 \setminus \mathcal{F}} (w_{BSH}(\neg x) \cdot w_{BSH}(\neg y))$$

Notice that both $w_{BSH}(x_i)$ and $BSH(x_i)$ express the importance of literal x_i . Since $BSH(x_i)$ is far more costly to compute than $w_{BSH}(x_i)$, *BSH* could only improve performance if it proves to be a better heuristic than w_{BSH} . So far, this claim only holds for random formulae.

Based on the assumption that $BSH(x_i)$ is better than $w_{BSH}(x_i)$, Dubois and Dequen [DD01] propose to iterate the above by using $BSH(x_i)$ instead of $w_{BSH}(x_i)$ for the second iteration to improve the accuracy of the heuristic. This can be repeated by using the $BSH(x_i)$ values of iteration i (with $i > 2$) to compute $BSH(x_i)$ of iteration $i + 1$. However, iterating *BSH* makes it much more costly. In general, performing only a single iteration is optimal in terms of solving time.

In [DD03] Dubois and Dequen offer a normalization procedure for *BSH* such that it weighs *all* newly created clauses - instead of only the binary ones. The normalization is required for fast performances on random k -SAT formula with $k > 3$. Backbone Search Renormalized Heuristic (in short *BSRH*) consists of two additional aspects: 1) Smaller clauses are more heavier using the γ_k weights, and 2) the w_{BSH} values to compute $BSH(x_i)$ are renormalized, by dividing them by the average weight of all literals in $\mathcal{F}[x_i = 1] \setminus \mathcal{F}$ - denoted by $\mu_{BSH}(x_i)$.

$$\mu_{BSH} := \frac{\sum_{C \in \mathcal{F}[x_i=1] \setminus \mathcal{F}} \sum_{x \in C} w_{BSH}(\neg x)}{\sum_{C \in \mathcal{F}[x_i=1] \setminus \mathcal{F}} |C|}$$

$$BSRH(x_i) := \sum_{C \in \mathcal{F}[x_i=1] \setminus \mathcal{F}} \left(\gamma_{|C|} \cdot \prod_{x \in C} \frac{w_{BSH}(\neg x)}{\mu_{BSH}(x_i)} \right)$$

Similar as *BSH*, *BSRH* can be iterated to improve accuracy.

5.3.1.4. Comparison of DIFF heuristics

Example 5.3.1. Consider the unsatisfiable 3-SAT formula $\mathcal{F}_{\text{heuristic}}$ below:

⁶Although the various descriptions of the heuristic [DD01, DD03] use a factor 2, the actual implementation of the heuristic in *knfs* (see <http://www.laria.u-picardie.fr/~dequen/sat/>) also uses factor 5.

$$\begin{aligned} \mathcal{F}_{\text{heuristic}} = & (\neg x_1 \vee x_2 \vee \neg x_6) \wedge (\neg x_2 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_7) \wedge \\ & (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \wedge \\ & (x_2 \vee x_3 \vee \neg x_4) \wedge (\neg x_2 \vee x_3 \vee x_5) \wedge (\neg x_2 \vee x_3 \vee \neg x_5) \wedge \\ & (x_1 \vee x_2 \vee x_6) \wedge (x_1 \vee x_2 \vee x_7) \wedge (x_2 \vee x_4 \vee \neg x_5) \wedge (x_2 \vee x_4 \vee x_5) \end{aligned}$$

Table 5.1 compares the heuristic values of look-aheads on $\mathcal{F}_{\text{heuristic}}$ using the three heuristics *CRW*, *WBH*, and *BSH*. Notice that *CRH* prefers x_2 as decision variable, while *WBH* and *BSH* prefer x_3 . The latter choice will result in a smaller search-tree. Between *WBH* and *BSH* the preferred ordering differs as well. For *BSH*, variables x_4 and x_5 are relatively more important than for *WBH*.

BSH is the most effective heuristic on random k -SAT formulae. The relative effectiveness of these heuristics on structured instances is not well studied yet.

Table 5.1. Comparison between the heuristic values of the look-ahead on the variables in $\mathcal{F}_{\text{heuristic}}$ using *CRH*, *WBH* and *BSH*. The product is used as MIXDIFF.

	$\#(x_i)$	$\#(\neg x_i)$	MIXDIFF(<i>CRH</i>)	MIXDIFF(<i>WBH</i>)	MIXDIFF(<i>BSH</i>)
x_1	4	4	$4 \cdot 4 = 16$	$24 \cdot 21 = 504$	$30 \cdot 27 = 810$
x_2	6	3	$3 \cdot 6 = 18$	$17 \cdot 33 = 516$	$20 \cdot 40 = 800$
x_3	3	4	$4 \cdot 3 = 12$	$30 \cdot 23 = 690$	$56 \cdot 36 = 2016$
x_4	4	3	$3 \cdot 4 = 12$	$21 \cdot 24 = 504$	$36 \cdot 36 = 1296$
x_5	2	2	$2 \cdot 2 = 4$	$15 \cdot 15 = 225$	$27 \cdot 27 = 729$
x_6	1	1	$1 \cdot 1 = 1$	$7 \cdot 7 = 49$	$12 \cdot 12 = 144$
x_7	1	1	$1 \cdot 1 = 1$	$10 \cdot 7 = 70$	$24 \cdot 12 = 288$

5.3.2. Direction heuristics

Various state-of-the-art satisfiability (SAT) solvers use *direction heuristics* to predict the sign of the decision variables: These heuristics choose, after the selection of the decision variable, which Boolean value is examined first. Direction heuristics are in theory very powerful: If always the correct Boolean value is chosen, satisfiable formulae would be solved without backtracking. Moreover, existence of perfect direction heuristics (computable in polynomial time) would prove that $\mathcal{P} = \mathcal{NP}$. These heuristics are also discussed in Section 8.9.

Although very powerful in theory, it is difficult to formulate effective direction heuristics in practice. Look-ahead SAT solvers even use complementary strategies to select the first branch - see Section 5.3.2.1.

Direction heuristics may bias the distribution of solutions in the search-tree. For instance, while using *march* or *kcdfs*, such a biased distribution is observed on hard random 3-SAT formulae. A biased distribution can be used to replace the DPLL *depth-first* search by a search that visits subtrees in order of increasing likelihood of containing a solution - see Section 5.3.2.2.

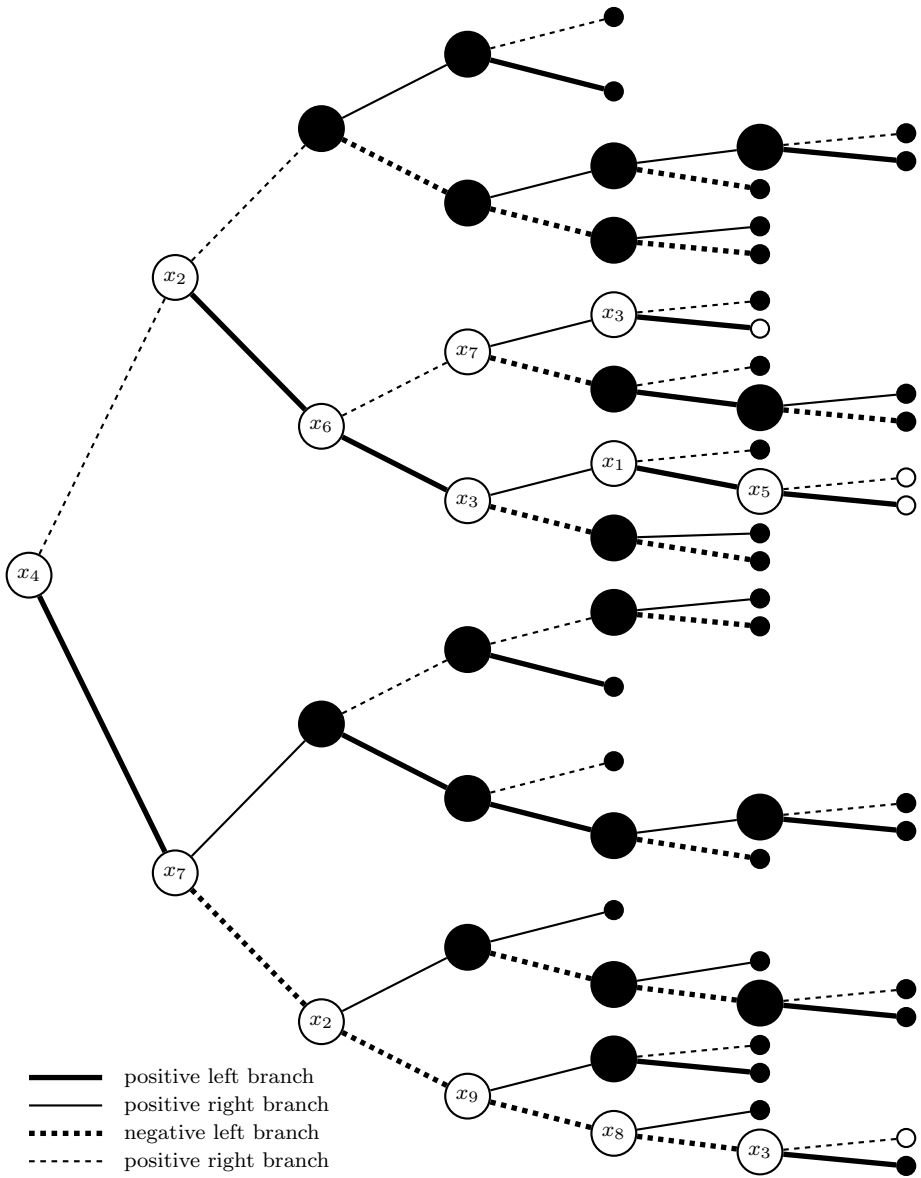


Figure 5.4. Complete binary search-tree (DPLL) for a formula with nine variables (x_1, \dots, x_9) . The decision variables are shown inside the internal nodes. A node is colored black if all child nodes are unsatisfiable, and white otherwise. The type of edge shows whether it is visited first (left branch), visited last (right branch), its decision variable is assigned to true (positive branch), or its decision variable is assigned to false (negative branch).

The search-tree of a DPLL-based SAT solver can be visualized as a binary search-tree, see Figure 5.4. This figure shows such a tree with decision variables drawn in the internal nodes. Edges show the type of each branch. We will refer to the *left branch* as the subformula that is visited first. Consequently, the *right branch* refers to the one examined later. A black leaf refers to an unsatisfiable dead end, while a white leaf indicates that a satisfying assignment has been found. An internal node is colored black in case both its children are black, and white otherwise. For instance, at depth 4 of this search-tree, 3 nodes are colored white. This means that at depth 4, 3 subtrees contain a solution.

5.3.2.1. Complementary strategies

A wide range of direction heuristics is used in look-ahead SAT solvers:

- **kcdfs**⁷ selects $\mathcal{F}[x_{\text{decision}} = 1]$ if x_{decision} occurs more frequently in \mathcal{F} than $\neg x_{\text{decision}}$. Otherwise it selects $\mathcal{F}[x_{\text{decision}} = 0]$.
- **march** selects $\mathcal{F}[x_{\text{decision}} = 1]$ if $\text{DIFF}(\mathcal{F}, \mathcal{F}[x_{\text{decision}} = 1])$ is smaller than $\text{DIFF}(\mathcal{F}, \mathcal{F}[x_{\text{decision}} = 0])$, and $\mathcal{F}[x_{\text{decision}} = 0]$ otherwise [HDvZvM04].
- **OKsolver** selects the subformula with the smallest probability that a random assignment will falsify a random formula of the same size [Kul02]. It prefers the minimum sum of

$$\sum_{k \geq 2} -|\mathcal{F}_k| \cdot \ln(1 - 2^{-k}) \quad (5.1)$$

- **posit** selects $\mathcal{F}[x_{\text{decision}} = 0]$ if x_{decision} occurs more often than $\neg x_{\text{decision}}$ in the shortest clauses of \mathcal{F} . Otherwise it selects $\mathcal{F}[x_{\text{decision}} = 1]$ [Fre95].
- **satz**⁸ does not use direction heuristics and starts with $\mathcal{F}[x_{\text{decision}} = 1]$.

These heuristics can be divided into two strategies:

A) *Estimate which subformula will require the least computational time.*

This strategy assumes that it is too hard to predict whether a subformula is satisfiable. Therefore, if both subformulae have the same expectation of containing a solution, you will find a solution faster by starting with the one that requires less computational time. From the above solvers, **posit** uses this strategy.

Notice that conflict-driven SAT solvers may prefer this strategy to focus on the most unsatisfiable subformula. This may result in an early conflict, and possibly a short conflict clause. However, since the current look-ahead SAT solvers do not add conflict clauses, this argument is not applicable for these solvers.

B) *Estimate which subformula has the highest probability of being satisfiable.*

If a formula is satisfiable, then it is expected that the subformula which is heuristically the most satisfiable will be your best bet. The other subformula may be unsatisfiable and - more importantly - a waste of time. The solvers **kcdfs**, **march**, and **OKsolver** use this strategy.

Generally these two strategies are complementary: The subformula which can be solved with relatively less computational effort is probably the one which is more

⁷version 2006

⁸version 2.15.2

constraint and therefore has a smaller probability being satisfiable. In practice, for example, satisfiable hard random k -SAT formulae are solved faster by using a direction heuristic based on strategy B , while on various structured satisfiable instances strategy A is more successful.

The *balancedness* of the DPLL search-tree could hint which is the best strategy to use: In a balanced search-tree, solving $\mathcal{F}[x_{\text{decision}} = 0]$ requires almost as much effort as solving $\mathcal{F}[x_{\text{decision}} = 1]$ for each decision variable x_{decision} . In this case, strategy A seems less appealing. Since look-ahead solvers produce a balanced search tree on hard random k -SAT formulae, strategy B would be preferred.

On the other hand, in an unbalanced search-tree one could get lucky using strategy A which would be therefore preferred. On many structured instances look-ahead SAT solvers produce an unbalanced search-tree - although the MIX-DIFF heuristic attempts to achieve a balanced search-tree.

Based on this explanation, `march_ks` uses both strategies: Strategy B is used (as implemented in `march_dl`) in case $\text{DIFF}(\mathcal{F}, \mathcal{F}[x = 0])$ and $\text{DIFF}(\mathcal{F}, \mathcal{F}[x = 1])$ are *comparable*. Otherwise, strategy A is used by preferring the complementary branch which would have been selected by `march_dl`. In `march_ks`, “comparable” is implemented as $0.1 \leq \frac{\text{DIFF}(\mathcal{F}, \mathcal{F}[x=0])}{\text{DIFF}(\mathcal{F}, \mathcal{F}[x=1])} \leq 10$.

5.3.2.2. Distribution of solutions

Given a family of benchmark instances, we can observe the effectiveness of the direction heuristic used in a certain solver on that particular family. The observation can be illustrated as follows: Consider the subtrees at depth d of the DPLL search-tree. These subtrees are denoted by T_i with $i = \{1, \dots, 2^d\}$ and are numbered in depth-first order. The histogram showing the number of formulae in the given family containing at least one solution in T_i is called a *solution distribution plot*.

Figures 5.5 and 5.6 show solution distribution plots at depth 12 for `march_dl` and `kcfnfs` respectively on 3000 random 3-SAT formulae with 350 variables and 1491 clauses (at phase transition density). The observed distribution is clearly biased for both solvers. The distribution observed using `march_dl` is more biased than the one using `kcfnfs`. Based on these plots, one could conclude that the direction heuristics used in `march_dl` are more effective for the instances at hand.

For both solvers, the number of formulae with at least one solution in T_i is highly related to the number of left branches that is required to reach T_i . With this in mind, the DPLL depth first search can be replaced [HvM06b] by a search strategy that, given a *jump depth*, visits the subtrees at that depth in the order of required left branches. Using jump depth 12, first subtree T_1 is visit (twelve left branches, zero right branches), followed by subtrees $T_2, T_3, T_5, T_9, \dots, T_{2049}$ (eleven left branches, one right branch), etc. Random satisfiable formulae of the same size are on average solved much faster using modified DPLL search.

Notice that for conflict-driven solvers this kind of solution distribution plots could not be made. The most important reason is that there is no clear left and right branch, because conflict clauses assign former decision variables to the complement of the preferred truth value. Another difficulty to construct these plots is the use of restarts in conflict-driven solvers. Finally, the use of lazy data-structures makes it very expensive to apply a direction heuristic that requires

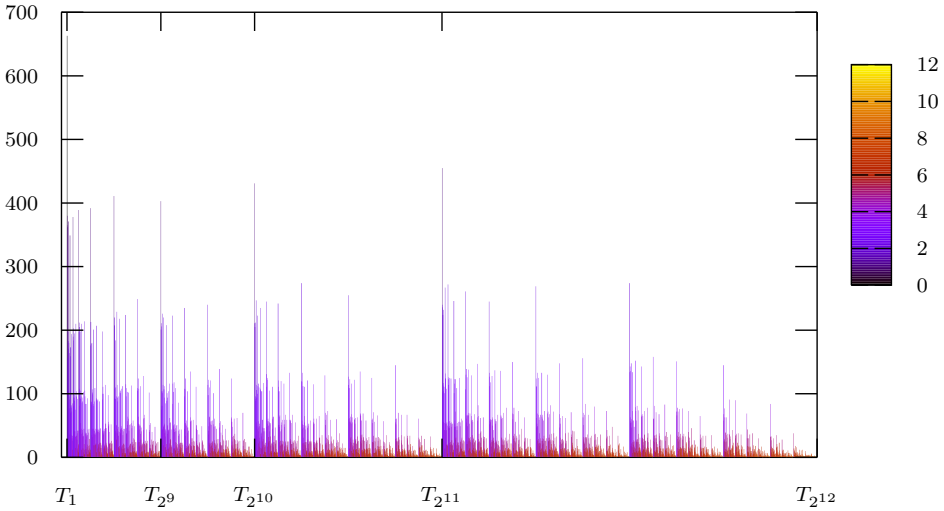


Figure 5.5. Solution distribution plot with `march_dl` showing for each subtree T_i at depth 12 the number of formulae that have at least one solution in that subtree. The experiments are based on 3000 random 3-SAT formulae with 350 variables and 1491 clauses. The colors show the number of right branches.

much statistical data.

5.3.3. Pre-selection heuristics

Overall performance can be gained or lost by performing look-ahead on a subset of the free variables in each node of the DPLL-tree: Gains are achieved by the reduction of computational costs, while losses are the result of either the inability of the *pre-selection heuristics* (heuristics that determine the set of variables to enter the look-ahead phase) to select effective decision variables or to predict candidate failed literals. When look-ahead is performed on a subset of the variables, only a subset of the failed literals is most likely detected. Depending on the formula, this could increase the size of the DPLL-tree. This section describes the two most commonly used pre-selection heuristics in look-ahead SAT solvers: *prop_z* and *clause reduction approximation*.

5.3.3.1. *prop_z*

Li [LA97a] developed the *prop_z* heuristic for his solver `satz`, which is also used in `kcdfs`. It pre-selects variables based on their occurrences in binary clauses. The *prop_z* heuristic is developed to perform faster on hard random k -SAT formulae. Near the root of the search-tree it pre-selects all free variables. From a certain depth on, it pre-selects only variables that occur both positive and negative in binary clauses. It always pre-selects a lower bound of 10 variables.

Although this heuristic is effective on hard random k -SAT formulae, its behavior on structured instances is not clear. On some benchmarks it will pre-select all

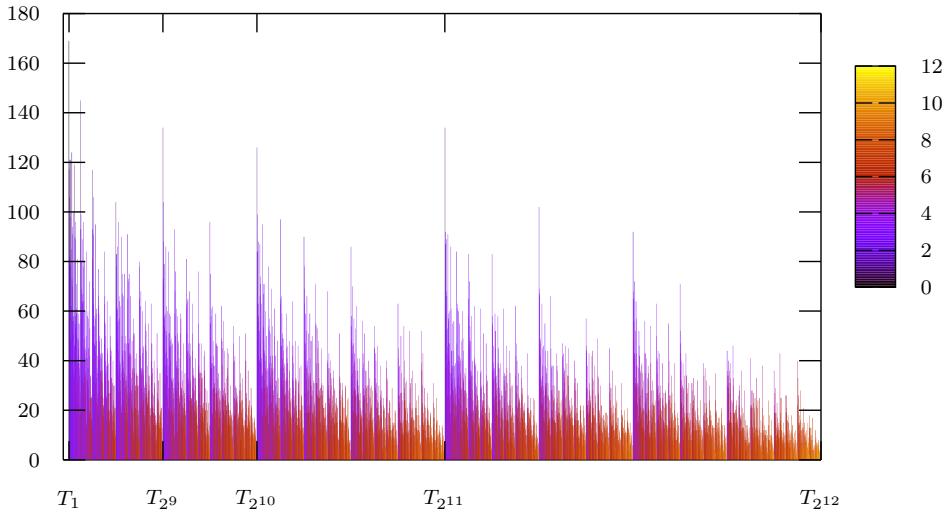


Figure 5.6. Solution distribution plot with `knfs` showing for each subtree T_i at depth 12 the number of formulae that have at least one solution in that subtree. The experiments are based on 3000 random 3-SAT formulae with 350 variables and 1491 clauses. The colors show the number of right branches.

free variables in all nodes of the search-tree, while on others it always pre-selects slightly more variables than the lower bound used.

Table 5.2. Average number of variables selected for the LOOKAHEAD procedure by $\text{RANK}_{10\%}$ and prop_z for 300 variables and 1275 clauses random 3-SAT problems.

depth	#freeVars	prop_z	depth	#freeVars	prop_z
1	298.24	298.24	11	264.55	26.81
2	296.52	296.52	12	260.53	21.55
3	294.92	293.89	13	256.79	19.80
4	292.44	292.21	14	253.28	19.24
5	288.60	280.04	15	249.96	19.16
6	285.36	252.14	16	246.77	19.28
7	281.68	192.82	17	243.68	19.57
8	277.54	125.13	18	240.68	19.97
9	273.17	71.51	19	237.73	20.46
10	268.76	40.65	20	234.82	20.97

5.3.3.2. Clause reduction approximation

In march the pre-selection heuristics are based on an approximation of a decision heuristic that counts the number of newly created clauses, called *clause reduction approximation (CRA)*. It uses the sizes of all clauses in the formula and is therefore more costly than prop_z . First, the set of variables to be assigned during look-ahead on x is approximated. All variables that occur with $\neg x$ in

binary clauses are used as approximated set. Second, if y_i occurs in this set, then all n -ary clauses in which $\neg y_i$ occurs will be reduced. This is an approximation of the number of newly created clauses, because some of the reduced ones will be satisfied. Third, the product is used as MIXDIFF. Let $\#_{>2}(x_i)$ denotes the number of occurrences of literal x_i in clauses with size > 2 . For all free variables CRA is computed as:

$$CRA(x) := \left(\sum_{x \vee y_i \in \mathcal{F}} \#_{>2}(\neg y_i) \right) \cdot \left(\sum_{\neg x \vee y_i \in \mathcal{F}} \#_{>2}(\neg y_i) \right) \quad (5.2)$$

Variables with the highest CRA scores are pre-selected for the LOOKAHEAD procedure. Once the CRA scores are computed, one has to determine how many variables should be pre-selected. The optimal number varies heavily from benchmark to benchmark and from node to node in the search-tree. Early march versions used a fraction of the original number of variables. Most versions used 10% of the number of variables, which is referred to as RANK_{10%}. Later versions use an adaptive heuristic discussed in the next paragraph.

5.3.3.3. Adaptive Ranking

As observed in [HDvZvM04], the optimal number of pre-selected variables is closely related to the number of detected failed literals: When relatively many failed literals were detected, a larger pre-selected set appeared optimal. Let $\#\text{failed}_i$ be the number of detected failed literals in node i . To exploit this correlation, the average number of detected failed literals is used as an indicator for the (maximum) size of the pre-selected set in node n (denoted by RANKADAPT _{n}):

$$\text{RANKADAPT}_n := L + \frac{S}{n} \sum_{i=1}^n \#\text{failed}_i \quad (5.3)$$

In the above, parameter L refers to the lower bound of RANKADAPT _{n} (namely when the average tends to zero) and S is a parameter modeling the importance of failed literals. `March_eq` uses $L := 5$ and $S := 7$ based on experiments on structured and random benchmarks. Notice that the above adaptive pre-selection heuristics are heavily influenced by the decision heuristics - which in turn are also affected by these heuristics.

Generally RANKADAPT select a subset of the free variables, but in some cases—due to many detected failed literals—all free variables are selected. It may occur that, during the LOOKAHEAD procedure, all selected variables are forced. In that case a new set of variables is pre-selected.

5.4. Additional Reasoning

Selection of decision variables by performing many look-aheads is very expensive compared to alternative decision heuristics. As such, this could hardly be considered as an advantage. However, since the decision heuristics are already costly, some additional reasoning is relatively cheap. This section presents various

techniques that can be added to the LOOKAHEAD procedure in order to improve overall performance.

Look-ahead on variables which will not result in a conflict appears only useful to determine which variable has the highest decision heuristic value. However, by applying some additional reasoning, look-ahead on some variables can also be used to reduce the formula (like failed literals). Look-ahead on the other variables can be used to add *resolvents* (learned clauses) to further constrain the formula. Three kinds of additional reasoning are used in look-ahead SAT solvers for these purposes: *Local learning* (Section 5.4.1), *autarky detection* (Section 5.4.2), and *double look-ahead* (Section 5.4.3).

5.4.1. Local learning

During the look-ahead on x , other variables y_i can be assigned by unit propagation. Some due to the presence of binary clauses $\neg x \vee (\neg)y_i$, called *direct implications*. Variables assigned by other clauses are called *indirect implications*. For those variables y_i that are assigned to true (or false) by a look-ahead on x though indirect implications, a binary clause $\neg x \vee y_i$ (or $\neg x \vee \neg y_i$, respectively) can be added to the formula. This is referred to as *local learning*. As the name suggests, these clauses are not globally valid and should therefore be removed while backtracking.

Example 5.4.1. Consider the formula $\mathcal{F}_{\text{learning}}$ below:

$$\mathcal{F}_{\text{learning}} := (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee x_3 \vee x_6) \wedge (x_1 \vee x_4 \vee \neg x_5) \wedge (x_1 \vee \neg x_6) \wedge (x_4 \vee x_5 \vee x_6) \wedge (x_5 \vee \neg x_6)$$

Some look-aheads on literals $(\neg)x_i$ will result in unit clauses, thereby assigning other variables. These assignments are listed below:

$$\begin{array}{ll} [x_1 := 0] \rightarrow \{x_3=1, x_6=0\} & [x_4 := 0] \rightarrow \{\} \\ [x_1 := 1] \rightarrow \{x_2=1, x_3=1, x_4=1\} & [x_4 := 1] \rightarrow \{\} \\ [x_2 := 0] \rightarrow \{x_1=0, x_3=1, x_6=0\} & [x_5 := 0] \rightarrow \{x_4=1, x_6=0\} \\ [x_2 := 1] \rightarrow \{\} & [x_5 := 1] \rightarrow \{\} \\ [x_3 := 0] \rightarrow \{\} & [x_6 := 0] \rightarrow \{\} \\ [x_3 := 1] \rightarrow \{\} & [x_6 := 1] \rightarrow \{x_1=1, x_2=1, x_3=1, x_4=1, x_5=1\} \end{array}$$

Eight of the above assignments follow from indirect implications. Therefore, these can be added to $\mathcal{F}_{\text{learning}}$:

$$\begin{array}{llll} x_1 \vee x_3 & \neg x_1 \vee x_4 & x_2 \vee \neg x_6 & x_4 \vee x_5 \\ \neg x_1 \vee x_3 & x_2 \vee \neg x_3 & x_3 \vee \neg x_6 & x_4 \vee \neg x_6 \end{array}$$

In the example above, the number of binary clauses that could be added by local learning is equal to the number of original clauses in $\mathcal{F}_{\text{learning}}$. In most real world examples, the number of local learned clauses is significantly larger than the number of original clauses. Therefore, adding all these binary clauses is generally not useful because it slows down propagation.

5.4.1.1. Necessary assignments

A cheap technique to reduce the size of the formula is detection of *necessary assignments*: If the look-ahead on x_i assigns x_j to true *and* the look-ahead on $\neg x_i$ assigns x_j to true then the assignment x_j to true is referred to as necessary. This technique requires only the short storage of implications. First, store all implications while performing look-ahead on x_i . Then continue with look-ahead on $\neg x_i$ and check for each implication whether it is in the stored list. All matches are necessary assignments. After look-ahead on $\neg x_i$ the stored implications can be removed.

In Example 5.4.1, variable x_3 is necessarily assigned to true because it is implied by both x_1 and $\neg x_1$. Necessary assignments can also be detected by adding local learned clauses: When these clauses are added, look-ahead on the complement of the necessary assignment will fail. Notice that by searching for necessary assignments, variables which are not in the pre-selected set can be forced to an assignment.

Searching for necessary assignments is a simplified technique of learning in the Stålmarck proof procedure [SS98]. This patented procedure learns the intersection of the new clauses in the reduced formulae (after look-ahead on x_i and $\neg x_i$). Necessary assignments are only the new unit clauses in this intersection. One could learn even more by adding clauses of size 2 and larger, but none of the current look-ahead SAT solvers adds these clauses because computing them is quite expensive. However, the HeerHugo SAT solver [GW00], inspired by the Stålmarck proof procedure, performs the “full” learning. The theory regarding these and other strengthenings of unit clause propagation are discussed in [Kul99a, Kul04].

5.4.1.2. Constraint resolvents

The concept of necessary assignments is not sufficient to detect all forced literals by using local learning. For example, after adding the learned clauses $\neg x_1 \vee x_4$ and $x_4 \vee x_5$ to $\mathcal{F}_{\text{learning}}$, look-ahead on $\neg x_4$ will fail, forcing x_4 to be assigned to true. Now, the question arises: Which local learned clauses should be added to detect all forced literals?

Local learned clauses are useful when the number of assigned variables is increased during look-ahead on the complement of one of its literals. For instance, given a formula with clauses $x_1 \vee x_2$ and $\neg x_2 \vee x_3$. The local learned clause $x_1 \vee x_3$ is not useful because look-ahead on $\neg x_1$ would already assign x_3 to true and look-ahead on $\neg x_3$ would already assign x_1 to true.

However, given a formula with clauses $x_1 \vee x_2$ and $x_1 \vee \neg x_2 \vee x_3$ then local learned clause $x_1 \vee x_3$ is useful: Only after adding this clause, look-ahead on $\neg x_3$ will assign x_1 to true. These useful local learned clauses are referred to as *constraint resolvents* [HDvZvM04].

In Example 5.4.1, all local learned clauses except $x_2 \vee \neg x_6$ are constraint resolvents. Yet, in practice, only a small subset of the local learned clauses are constraint resolvents. In Section 5.5.1 a technique is explained to efficiently detect constraint resolvents.

Constraint resolvents and heuristics. Pre-selection heuristics rank variables based on their occurrences in binary clauses - see Section 5.3.3. Addition of constraint resolvents (or local learned clauses in general) will increase the occurrences in binary clauses of variables in the pre-selected set in particular. So, once variables are pre-selected, their rank will rise, increasing the chances of being pre-selected again in a next node. Therefore, it becomes harder for “new” variables to enter the look-ahead phase.

5.4.2. Autarky Reasoning

An *autarky* (or autark assignment) is a partial assignment φ that satisfies all clauses that are “touched” by φ . So, all satisfying assignments are autark assignments. Autarkies that do not satisfy all clauses can be used to reduce the size of the formula: Let $\mathcal{F}_{\text{touched}}$ be the clauses in \mathcal{F} that are satisfied by an autarky. The remaining clauses $\mathcal{F}^* := \mathcal{F} \setminus \mathcal{F}_{\text{touched}}$ are satisfiability equivalent with \mathcal{F} - see Chapter 14. So, if we detect an autark assignment, we can reduce \mathcal{F} by removing all clauses in $\mathcal{F}_{\text{touched}}$.

5.4.2.1. Pure literals

The smallest example of an autarky is a *pure literal*: A literal whose negation does not occur in the formula. Consider the example formula:

$$\mathcal{F}_{\text{pure literal}} := (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$$

Variable x_2 occurs only negated in this formula - making $\neg x_2$ a pure literal. Assigning x_2 to false will therefore only satisfy clauses. Under this assignment the only clause left is $(\neg x_1 \vee x_3)$. Now both $\neg x_1$ and x_3 are pure literals. So by assigning a pure literal to true could make other literals pure.

5.4.2.2. Autarky detection by look-ahead

Look-aheads can also be used to detect autarkies. Whether a look-ahead resulted in an autark assignment requires a check that all reduced clauses became satisfied. Recall that the *CRH* and the *BSH* look-ahead evaluation heuristics count (and weight, respectively) the newly created clauses. While using these heuristics, an autarky detection check can be computed efficiently: All reduced clauses are satisfied if and only if their heuristic value is 0.

Example 5.4.2. Consider the formula $\mathcal{F}_{\text{autarky}}$ below.

$$\mathcal{F}_{\text{autarky}} := (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_5) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_3 \vee x_4) \wedge (\neg x_2 \vee \neg x_3 \vee \neg x_4) \wedge (x_2 \vee x_4 \vee \neg x_5)$$

The heuristic values of the *CRH* look-ahead evaluation heuristic (in this case the number of newly created binary clauses) of all possible look-aheads on $\mathcal{F}_{\text{autarky}}$ are shown in Table 5.3. Both the look-aheads on $\neg x_3$ and $\neg x_5$ result in a heuristic value of 0. The first represents the autark assignment $\varphi = \{x_3 = 0, x_4 = 1\}$, and the second represents the autarky assignment $\varphi = \{x_1 = x_2 = x_5 = 0\}$.

Table 5.3. Number of new binary clauses after a look-ahead on x_i on $\mathcal{F}_{\text{autarky}}$. For this formula these values equal $CRH(x_i)$ because the largest clause has length 3.

	x_1	$\neg x_1$	x_2	$\neg x_2$	x_3	$\neg x_3$	x_4	$\neg x_4$	x_5	$\neg x_5$
$CRH(x_i)$	2	2	1	2	2	0	1	2	1	0

In general, look-ahead SAT solvers do not check whether a look-ahead satisfies all remaining clauses - thereby detecting a satisfying assignment. By detecting autarkies, satisfying assignments will also be detected as soon as they appear.

5.4.2.3. Look-ahead autarky resolvents

In case a look-ahead satisfies all but one of the reduced clauses, we have almost detected an autarky. In Example 5.4.2 the look-ahead on x_2 , x_4 , and x_5 created only one clause that was not satisfied: $\neg x_3 \vee \neg x_4$, $\neg x_2 \vee \neg x_3$, and $x_2 \vee x_4$ respectively. Due to these binary clauses, none of the variables is forced to a value.

However, we know that if such a particular clause were satisfied, we would have detected an autarky. This knowledge can be added to the formula: In case of the look-ahead on x_2 , if $\neg x_3 \vee \neg x_4$ is satisfied x_2 is true. So $\neg x_3 \rightarrow x_2$ and $\neg x_4 \rightarrow x_2$. The binary clauses of these implications, $x_3 \vee x_2$ and $x_4 \vee x_2$, are referred to as *look-ahead autarky resolvents*. Recall that the look-ahead on $\neg x_3$ in the autarky $\varphi = \{x_3 = 0, x_4 = 1\}$. After the addition of $x_3 \vee x_2$, the same look-ahead will result in the autarky $\varphi = \{x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 1, x_5 = 1\}$.

Look-ahead autarky resolvents can be added to the formula to further constrain it. Generally, let C be the only reduced clause which is not satisfied after the look-ahead on x . For all literals l_i in C we can add to the formula a look-ahead autarky resolvent $x \vee \neg l_i$. More details about look-ahead autarky resolvents are described in Section 14.13.2 and [Kul99c, Kul99b].

Look-ahead autarky resolvents and heuristics. Although look-ahead autarky resolvents further constrain a formula, their addition could contribute to a significant drop in performance of a solver. The reason for this paradox can be found in the effect of look-ahead autarky resolvents on the heuristics: According to difference heuristics, variables are important if they create many new clauses. Therefore, variables on which look-ahead results in only a single new clause should be considered unimportant. However, the heuristic values of *WBH*, *BSH*, and *CRA* increases after adding these look-ahead autarky resolvents. This makes it more likely that unimportant variables are chosen as decision variable. Therefore, it is useful to neglect these clauses while computing the heuristics.

5.4.3. Double look-ahead

Due to the design of MIXDIFF heuristics in look-ahead SAT solvers, unbalanced variables are rarely selected as decision variables. To compensate for this, Li developed the DOUBLELOOK procedure [Li99]. This procedure performs look-aheads on a second level of propagation (on the reduced formula after a look-ahead) to increase the number of detected forced literals.

We refer to the *double look-ahead* on x as calling the DOUBLELOOK procedure on the reduced formula $\mathcal{F}[x = 1]$. A double look-ahead on x is called successful if the DOUBLELOOK procedure detects that $\mathcal{F}[x = 1]$ is unsatisfiable. Otherwise it is called unsuccessful. Similar to a failed look-ahead on x , a successful double look-ahead on x detects a conflict in $\mathcal{F}[x = 1]$ and thereby forces x to be assigned to false.

Although double look-aheads are expensive, they can be a relatively efficient method to detect some forced literals in practice. In order to reduce the solving time, many double look-aheads must be successful. Prediction of the success of a double look-ahead is therefore essential.

Double look-ahead heuristics. The heuristics regarding double look-aheads focus on the success predictor. Li suggests to use the number of newly created binary clauses as an effective predictor [Li99]. If the number of newly created binary clauses during look-ahead on x (denoted by $|\mathcal{F}[x = 1]_2 \setminus \mathcal{F}|$) is larger than a certain parameter (called Δ_{trigger}), then the DOUBLELOOK procedure should be triggered. All double look-ahead heuristics are concerned with the optimal value for Δ_{trigger} .

The first implementation of the DOUBLELOOK procedure used a static value for Δ_{trigger} . Li implemented $\Delta_{\text{trigger}} := 65$ in his solver *satz*. This magic constant is based on experiments on hard random 3-SAT formulae. Although these instances can be solved much faster using this setting, on structured instances it frequently results in the call of too many double look-aheads, which will reduce overall performance.

Another static implementation for Δ_{trigger} is found in *kcncfs* by Dubois and Dequen. They use $\Delta_{\text{trigger}} := 0.17 \cdot n$, with n referring to the original number of variables. This setting also arises from experiments on random 3-SAT formulae. Especially on random and structured instances with large n , better results are obtained compared to $\Delta_{\text{trigger}} := 65$.

A first dynamic heuristic was developed by Li for a later version of *satz*. It initializes $\Delta_{\text{trigger}} := 0.17 \cdot n$. If a double look-ahead on x is unsuccessful, Δ_{trigger} is updated to $\Delta_{\text{trigger}} := |\mathcal{F}[x = 1]_2 \setminus \mathcal{F}|$. The motivation for the update is as follows: Assuming the number of newly created binary clauses is an effective predictor for success of a double look-ahead, and because the double look-ahead on x was unsuccessful, Δ_{trigger} should be at least $|\mathcal{F}[x = 1]_2 \setminus \mathcal{F}|$. After a successful DoubleLook call, Δ_{trigger} is reset to $\Delta_{\text{trigger}} = 0.17 \cdot n$. In practice, this dynamic heuristic "turns off" the DOUBLELOOK procedure on most structured instances. The procedure is only rarely triggered due to the first update rule. The performance on these instances is improved to back to normal (i.e. not calling the DOUBLELOOK at all).

A second dynamic heuristic was developed by Heule and Van Maaren for their solver *march_dl* [HvM07]. It initializes $\Delta_{\text{trigger}} := 0$. Like the dynamic heuristic in *satz*, it updates $\Delta_{\text{trigger}} = |\mathcal{F}[x = 1]_2 \setminus \mathcal{F}|$ after an unsuccessful double look-ahead on x . The important difference is that Δ_{trigger} is not decreased after a successful doublelook, but Δ_{trigger} is slightly reduced after each look-ahead. Therefore, double look-aheads are performed once in a while. On random instances this dynamic heuristic yields comparable performances on random formulae. However, the use

of the DOUBLELOOK procedure with this heuristic improves the performance on many structured instances as well.

5.4.3.1. Double look-ahead resolvents

An unsuccessful double look-ahead is generally not a total waste of time. For example, the look-ahead on x_i triggers the DOUBLELOOK procedure. On the second level of propagation three failed literals are detected: x_r , $\neg x_s$, and x_t . Thus we learn $x_i \rightarrow \neg x_r$, $x_i \rightarrow x_s$, and $x_i \rightarrow \neg x_t$ - equivalent to the binary clauses $\neg x_i \vee \neg x_r$, $\neg x_i \vee x_s$, $\neg x_i \vee \neg x_t$. Since the DOUBLELOOK procedure performs additional look-aheads on free variables in $\mathcal{F}[x_i = 1]$, we know that \mathcal{F} does not contain these binary clauses - otherwise they would have been assigned already.

We refer to these binary clauses as *double look-ahead resolvents*. In case a double look-ahead is unsuccessful, double look-ahead resolvents can be added to \mathcal{F} to further constrain the formula. On the other hand, a successful double look-ahead on x_i will force x_i to false, thereby satisfying all these resolvents.

Adding double look-ahead resolvents can be useful to reduce overall costs caused by the DOUBLELOOK procedure. If a look-ahead on x_i triggers this procedure and appears to be unsuccessful, it is likely that it will trigger it again in the next node of the search-tree. By storing the failed literals at the second level of propagation as resolvents, the DOUBLELOOK procedure does not have to perform costly look-aheads to detect them again.

Double look-ahead resolvents and heuristics. Since the DOUBLELOOK procedure is triggered when a look-ahead on literal $(\neg)x_i$ creates many new binary clauses, the DIFF values for those literals is relatively high. Adding double look-ahead resolvents will further boost these DIFF values. This increases the chance that variables are selected as decision variable because the DIFF of either its positive or negative literal is very large. Concluding: By adding double look-ahead resolvents, decision heuristics may select decision variables yielding a more unbalanced search-tree.

5.5. Eager Data-Structures

Unit propagation is the most costly part of state-of-the-art complete SAT solvers. Within conflict-driven solvers, the computational costs of unit propagation is reduced by using lazy data-structures such as 2-literal watch pointers. Yet, for look-ahead SAT solvers these data-structures are not useful: The costs of unit propagation in these solvers is concentrated in the LOOKAHEAD procedure due to the numerous look-aheads. To compute the difference heuristics (as presented in Section 5.3.1), one requires the sizes of the reduced clauses. By using lazy data-structures, these sizes cannot be computed cheaply. However, by using *eager data-structures* the unit propagation costs can be reduced. The use of eager data-structures originates from the Böhm SAT solver [BS96]. This section offers three techniques to reduce the costs of unit propagation:

- **Efficient storage of binary clauses:** Binary clauses can be stored such that they require only half the memory compared to conventional storage.

In general, structured instances consists mostly of binary clauses. This significantly reduces the storage of the formula. Such a way of storage reduces the cost of unit propagation and makes it possible to cheaply detect constraint resolvents - see Section 5.5.1

- **Removal of inactive clauses:** Many of the original clauses become inactive (satisfied) down in the search-tree. Look-ahead can be performed faster if these clauses are removed from the data-structures - see Section 5.5.2
- **Tree based look-ahead:** Many propagations made during the look-ahead phase are redundant. A technique called tree-based look-ahead reduces this redundancy using implication trees - see Section 5.5.3.

5.5.1. Binary implication arrays

Generally, clauses are stored in a clause database together with a look-up table for each literal in which clauses it occurs. Such a storage is relatively cheap for binary clauses: Instead of storing the clauses, one could add the implications of assigning a literal to true directly in the “look-up table” [PH02]. We refer to such a storage as *binary implication arrays*. Notice that the latter data-structure requires only half the space compared to the former one. Figure 5.7 shows a graphical example of both data-structures for four binary clauses. Storing clauses in separate binary and n-ary data-structures reduced the computational costs of unit propagation: 1) Cache performance is improved because of the cheaper storage, and 2) No look-up is required for binary clauses when they are stored in binary implication arrays.

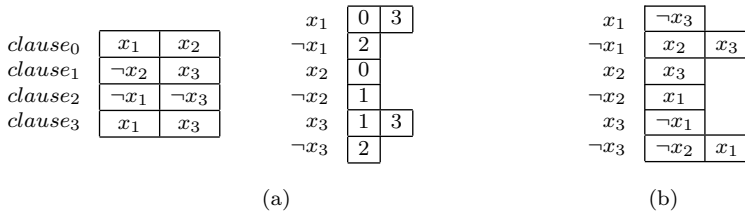


Figure 5.7. (a) A structure with binary clauses stored in a clause database, and literal look-up arrays containing the indices that point to the clauses in which they occur. (b) Binary clauses stored in implication arrays.

Besides the cheaper storage, splitting a formula in binary clauses (\mathcal{F}_2) and n -ary clauses ($\mathcal{F}_{>2}$) is also useful to efficiently detect constraint resolvents: Recall that constraint resolvents consist of the complement of the look-ahead literal and a literal that only occurs in a unit clause that originates from an n -ary clause. By performing unit propagation in such a way that unit clauses resulting from clauses in \mathcal{F}_2 are always preferred above those in $\mathcal{F}_{>2}$, the latter can be used to construct constraint resolvents. Algorithm 5.4 shows this *binary clause preferred (BCP) unit propagation* including the addition of constraint resolvents. Notice that this procedure returns both an expanded formula with learned constraint resolvents ($\mathcal{F}_2 \cup \mathcal{F}_{>2}$) and a reduced formula by the look-ahead ($\mathcal{F}[x = 1]$).

Algorithm 5.4 BCPUNITPROPAGATION (formula \mathcal{F} , variable x)

```

1:  $\varphi := \{x \leftarrow 1\}$ 
2: while empty clause  $\perp \notin \varphi \circ \mathcal{F}_2$  and a unit clause  $\in \varphi \circ \mathcal{F}_2$  do
3:   while empty clause  $\perp \notin \varphi \circ \mathcal{F}_2$  and unit clause  $\{y\} \in \varphi \circ \mathcal{F}_2$  do
4:      $\varphi := \varphi \cup \{y \leftarrow 1\}$ 
5:   end while
6:   if empty clause  $\perp \notin \varphi \circ \mathcal{F}_{>2}$  and unit clause  $\{y\} \in \varphi \circ \mathcal{F}_{>2}$  then
7:      $\mathcal{F}_2 := \mathcal{F}_2 \cup \{\neg x \vee y\}$ 
8:   end if
9: end while
10: return  $\mathcal{F}_2 \cup \mathcal{F}_{>2}, \varphi \circ \mathcal{F}$ 

```

5.5.2. Removal of inactive clauses

The presence of inactive clauses increases the computational costs of unit propagation during the LOOKAHEAD procedure. Two important causes can be observed: First, the larger the number of clauses considered during a look-ahead, the poorer the performance of the cache. Second, if both active and inactive clauses occur in the active data-structure during the look-ahead, a check is required to determine the status of every clause. Removal of inactive clauses from the active data-structure prevents these unfavorable effects from taking place.

All satisfied clauses are clearly inactive clauses. In case clauses are stored in separate binary and n -ary data-structures (see Section 5.5.1), then also clauses become inactive if they are represented in both data-structures: If an n -ary clause becomes a binary one and is added to the binary clause data-structure, it can be removed from the n -ary data-structure.

5.5.3. Tree-based look-ahead

Suppose the LOOKAHEAD procedure is about to perform look-ahead on the free variables x_1 , x_2 , and x_3 on a formula that contains the binary clauses $x_1 \vee \neg x_2$ and $x_1 \vee \neg x_3$. Obviously, the look-ahead on x_1 will assign all variables that are forced by $x_1 = 1$. Also, due to the binary clauses, the look-ahead on x_2 and x_3 will assign these variables (amongst others). Using the result of the look-ahead on x_1 , the look-aheads on x_2 and x_3 can be performed more efficiently.

Generally, suppose that two look-ahead literals share a certain implication. In this simple case, we could propagate the shared implication first, followed by a propagation of one of the look-ahead literals, backtracking the latter, then propagating the other look-ahead literal and finally backtracking to the initial state. This way, the shared implication has been propagated only once.

Figure 5.8 shows this example graphically. The implications (from the binary clauses) among x_1 , x_2 and x_3 form a small tree. Some thought reveals that this process, when applied recursively, could work for arbitrary trees. Based on this idea - at the start of each look-ahead phase - trees can be constructed from the implications between the literals selected for look-ahead, in such a way that each literal occurs in exactly one tree. By recursively visiting these trees, the LOOKAHEAD procedure is more efficient. Of course, the more dense the implication graph which arises from the binary clauses, the more possibilities are available to form trees. Adding all sorts of resolvents will in many cases be an important catalyst for the effectiveness of this method.

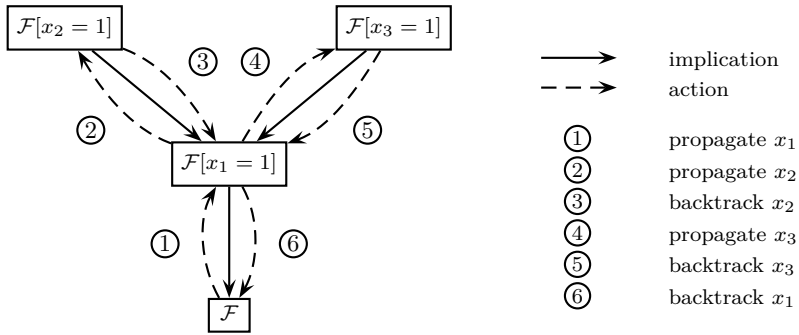


Figure 5.8. Graphical form of an implication tree with corresponding actions.

References

- [BK92] Michael Buro and Hans Kleine Büning. Report on a SAT competition, 1992.
- [BS96] Max Böhm and Ewald Speckenmeyer. A fast parallel SAT-solver - efficient workload balancing. *Ann. Math. Artif. Intell.*, 17(3-4):381–400, 1996.
- [CKS95] J. M. Crawford, M. J. Kearns, and R. E. Schapire. The minimal disagreement parity problem as a hard satisfiability problem, 1995.
- [DABC93] Olivier Dubois, Pascal André, Yacine Boufkhad, and Jacques Carlier. SAT versus UNSAT. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 11-13, 1993*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 415–436. DIMACS/AMS, 1993.
- [DD01] Olivier Dubois and Gilles Dequen. A backbone-search heuristic for efficient solving of hard 3-SAT formulae. In Bernhard Nebel, editor, *IJCAI*, pages 248–253. Morgan Kaufmann, 2001.
- [DD03] Gilles Dequen and Olivier Dubois. kcnfs: An efficient solver for random k -SAT formulae. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 486–501. Springer, 2003.
- [DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.
- [Fre95] Jon William Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, University of Pennsylvania, Philadelphia, PA, USA, 1995.
- [Fre96] Jon W. Freeman. Hard random 3-SAT problems and the davis-putnam procedure. *Artificial Intelligence*, 81(1):183 – 198, 1996. *Frontiers in Problem Solving: Phase Transitions and Complexity*.
- [GW00] J. F. Groote and J. P. Warners. The propositional formula checker heerhugo. *Journal of Automated Reasoning*, 24:101–125, 2000.

- [HDvZvM04] Marijn J. H. Heule, Mark Dufour, Joris E. van Zwieten, and Hans van Maaren. March_eq: Implementing additional reasoning into an efficient look-ahead SAT solver. In Holger H. Hoos and David G. Mitchell, editors, *SAT (Selected Papers)*, volume 3542 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 2004.
- [Her06] Paul Herwig. Decomposing satisfiability problems. Master’s thesis, TU Delft, 2006.
- [HvM06a] Marijn J. H. Heule and Hans van Maaren. March_dl: Adding adaptive heuristics and a new branching strategy. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:47–59, 2006.
- [HvM06b] Marijn J. H. Heule and Hans van Maaren. Whose side are you on? finding solutions in a biased search-tree. Technical report, Proceedings of Guangzhou Symposium on Satisfiability In Logic-Based Modeling, 2006.
- [HvM07] Marijn J. H. Heule and Hans van Maaren. Effective incorporation of double look-ahead procedures. In Joao Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing - SAT 2007*, volume 4501 of *Lecture Notes in Computer Science*, pages 258–271. Springer, 2007.
- [KL99] Oliver Kullmann and H. Luckhardt. *Algorithms for SAT/TAUT decision based on various measures*, 1999. Preprint, 71 pages, available on <http://cs.swan.ac.uk/~csoliver/Artikel/TAUT.ps>.
- [Kul99a] Oliver Kullmann. Investigating a general hierarchy of polynomially decidable classes of CNF’s based on short tree-like resolution proofs. Technical Report TR99-041, Electronic Colloquium on Computational Complexity (ECCC), October 1999.
- [Kul99b] Oliver Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, 223(1-2):1–72, July 1999.
- [Kul99c] Oliver Kullmann. On a generalization of extended resolution. *Discrete Applied Mathematics*, 96-97(1-3):149–176, 1999.
- [Kul02] Oliver Kullmann. Investigating the behaviour of a SAT solver on random formulas. Technical Report CSR 23-2002, University of Wales Swansea, Computer Science Report Series (<http://www-compsci.swan.ac.uk/reports/2002.html>), October 2002. 119 pages.
- [Kul04] Oliver Kullmann. Upper and lower bounds on the complexity of generalised resolution and generalised constraint satisfaction problems. *Annals of Mathematics and Artificial Intelligence*, 40(3-4):303–352, March 2004.
- [LA97a] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *IJCAI (1)*, pages 366–371, 1997.
- [LA97b] Chu Min Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In Gert Smolka, editor, *CP*, volume 1330 of *Lecture Notes in Computer Science*, pages 341–355. Springer, 1997.
- [Li99] Chu Min Li. A constraint-based approach to narrow search trees for satisfiability. *Information processing letters*, 71(2):75–80, 1999.
- [Li00] Chu Min Li. Integrating equivalency reasoning into davis-putnam

- procedure. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 291–296. AAAI Press / The MIT Press, 2000.
- [MZK⁺99] Rémi Monasson, Riccardo Zecchina, Scott Kirkpatrick, Bart Selman, and Lidror Troyansky. Determining computational complexity from characteristic 'phase transitions'. *Nature*, 400:133–137, 1999.
- [PH02] S. Pilarski and G. Hu. Speeding up sat for eda. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 1081, Washington, DC, USA, 2002. IEEE Computer Society.
- [SS98] Mary Sheeran and Gunnar Stålmarck. A tutorial on stålmarck's proof procedure for propositional logic. In *FMCAD '98: Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design*, pages 82–99, London, UK, 1998. Springer-Verlag.

Chapter 6

Incomplete Algorithms

Henry Kautz, Ashish Sabharwal, and Bart Selman

An *incomplete* method for solving the propositional satisfiability problem (or a general constraint satisfaction problem) is one that does not provide the guarantee that it will eventually either report a satisfying assignment or declare that the given formula is unsatisfiable. In practice, most such methods are biased towards the satisfiable side: they are typically run with a pre-set resource limit, after which they either produce a valid solution or report failure; they never declare the formula to be unsatisfiable. These are the kind of algorithms we will discuss in this chapter. In complexity theory terms, such algorithms are referred to as having one-sided error. In principle, an incomplete algorithm could instead be biased towards the unsatisfiable side, always providing proofs of unsatisfiability but failing to find solutions to some satisfiable instances, or be incomplete with respect to both satisfiable and unsatisfiable instances (and thus have two-sided error).

Unlike systematic solvers often based on an exhaustive branching and backtracking search, incomplete methods are generally based on *stochastic local search*, sometimes referred to as SLS. On problems from a variety of domains, such incomplete methods for SAT can significantly outperform DPLL-based methods. Since the early 1990's, there has been a tremendous amount of research on designing, understanding, and improving local search methods for SAT.¹ There have also been attempts at hybrid approaches that explore combining ideas from DPLL methods and local search techniques [e.g. HLDV02, MSG96, Pre01, RD96]. We cannot do justice to all recent research in local search solvers for SAT, and will instead try to provide a brief overview and touch upon some interesting details. The interested reader is encouraged to further explore the area through some of the nearly a hundred publications we cite along the way.

We begin the chapter by discussing two methods that played a key role in the success of local search for satisfiability, namely **GSAT** [SLM92] and **Walksat** [SKC93]. We will then discuss some extensions of these ideas, in particular clause weighting schemes which have made local search solvers highly competitive [CI96,

¹Various works [APSS05, CI96, FCS97, GW93, GM94, Gu92, GPFW97, HK05, Hoo99, Hoo02, HS04, KS94, Kon94, LWZ07, LSB03, MSK97, Mor93, PW96, PTS07, RF93, SS01, Spe93, TPBF04, WW00], and others.

Fra97, HTH02, ITA⁺06, Mor93, SLM92, TPBF04], and explore alternative techniques based on the discrete Lagrangian method [SSH01, SW98, WW99, WW00]. We will close this chapter with a discussion of the phase transition phenomenon in random k -SAT [CA93, KS94, MSL92] which played a critical role in the development of incomplete methods for SAT in the 1990s, and mention a relatively new incomplete technique for SAT called Survey Propagation [MPZ02].

These ideas lie at the core of most of the local search based competitive incomplete SAT solvers out there today, and have been refined and implemented in a number of very successful prototypes including `adaptg2wsat+`, `AdaptNovelty`, `gNovelty+`, `R+AdaptNovelty+`, `SAPS`, `UBCSAT`, `UnitWalk`, and `Walksat`. Rather than going into the details of each individual solver, we hope to present the broader computer science concepts that form the backbone of these solvers and to provide the reader with enough background and references to further explore the intricacies if desired.

We note that there are other exciting incomplete solution methodologies, such as those based on translation to Integer Programming [Hoo88, KKRR90], Evolutionary and Genetic Algorithms [DJW02, ES03, GMR02, LSH06], and Finite Learning Automata [GB07], that we will not discuss here. There has also been work on formally analyzing local search methods, yielding some of the best $o(2^n)$ time algorithms for SAT. For instance, the expected running time, ignoring polynomial factors, of a simple local search method with restarts after every $3n$ “flips” has been shown to be $(2 \cdot (k - 1)/k)^n$ for k -SAT [Sch99, Sch02], which yields a complexity of $(4/3)^n$ for 3-SAT. This result has been derandomized to yield a deterministic algorithm with complexity 1.481^n up to polynomial factors [DGH⁺02]. We refer the reader to Part 1, Chapter 12 of this Handbook for a detailed discussion of worst-case upper bounds for k -SAT.

For the discussion in the rest of this chapter, it will be illustrative to think of a propositional formula F with n variables and m clauses as creating a discrete manifold or *landscape* in the space $\{0, 1\}^n \times \{0, 1, \dots, m\}$. The 2^n truth assignments to the variables of F correspond to the points in $\{0, 1\}^n$, and the “height” of each such point, a number in $\{0, 1, \dots, m\}$, corresponds to the number of clauses of F that are violated by this truth assignment. The solutions or satisfying assignments for F are precisely the points in this landscape with height zero, and thus correspond to the global minima of this landscape, or, equivalently, the global minima of the function that maps each point in $\{0, 1\}^n$ to its height. The search problem for SAT is then a search for a global minimum in this implicitly defined exponential-size landscape. Clearly, if the landscape did not have any local minima, a greedy descent would provide an effective search method. All interesting formulas, however, do have local minima—the main challenge and opportunity for the designers of local search methods.

This landscape view also leads one naturally to the problem of *maximum satisfiability* or MAX-SAT: Given a formula F , find a truth assignment that satisfies the most number of clauses possible. Solutions to the MAX-SAT problem are, again, precisely the global minima of the corresponding landscape, only that these global minima may not have height zero. Most of the incomplete methods we will discuss in this chapter, especially those based on local search, can also work as a solution approach for the MAX-SAT problem, by providing the “best found”

truth assignment (i.e., one with the lowest observed height) upon termination. Of course, while it is easy to test whether the best found truth assignment is a solution to a SAT instance, it is NP-hard to perform the same test for a MAX-SAT instance. Thus, this approach only provides a heuristic algorithm for MAX-SAT, with a two-sided error. For further details on this problem, we refer the reader to Part 2, Chapter 19 of this Handbook.

6.1. Greedy Search and Focused Random Walk

The original impetus for trying a local search method on the satisfiability problem was the successful application of such methods for finding solutions to large N -queens instances, first using a connectionist system by Adorf and Johnston [AJ90], and then using greedy local search by Minton et al. [MJPL90]. It was originally assumed that this success simply indicated that N -queens was an easy problem, and researchers felt that such techniques would fail in practice for SAT and other more intricate problems. In particular, it was believed that local search methods would easily get stuck in local minima, with a few clauses remaining unsatisfied. Experiments with the solver **GSAT** showed, however, that certain local search strategies often do reach global minima, in many cases much faster than systematic search methods.

GSAT is based on a randomized local search technique proposed for combinatorial optimization by Lin and Kernighan [LK73] and Papadimitriou and Steiglitz [PS82]. The basic **GSAT** procedure, introduced by Selman et al. [SLM92] and described here as Algorithm 6.1, starts with a randomly generated truth assignment for all variables. It then greedily changes (‘flips’) the truth assignment of the variable that leads to the greatest decrease in the total number of unsatisfied clauses. The *neighborhood* of the current truth assignment, thus, is the set of n truth assignments each of which differs from the current one in the value of exactly one variable. Such flips are repeated until either a satisfying assignment is found or a pre-set maximum number of flips (**MAX-FLIPS**) is reached. This process is repeated as needed, up to a maximum of **MAX-TRIES** times.

Selman et al. showed that **GSAT** substantially outperformed even the best backtracking search procedures of the time on various classes of formulas, including randomly generated formulas and SAT encodings of graph coloring instances [JAMS91]. The search of **GSAT** typically begins with a rapid greedy descent towards a better truth assignment (i.e., one with a lower height), followed by long sequences of “sideways” moves. Sideways moves are moves that do not increase or decrease the total number of unsatisfied clauses. In the landscape corresponding to the formula, each collection of truth assignments that are connected together by a sequence of possible sideways moves is referred to as a *plateau*. Experiments indicate that on many formulas, **GSAT** spends most of its time on plateaus, transitioning from one plateau to another every so often. Interestingly, Frank et al. [FCS97] observed that in practice, almost all plateaus do have so-called “exits” that lead to another plateau with a lower number of unsatisfied clauses. Intuitively, in a very high dimensional search space such as the space of a 10,000 variable formula, it is very rare to encounter local minima, which are plateaus from where there is no local move that decreases the number

Algorithm 6.1: GSAT (F)

```

Input      : A CNF formula  $F$ 
Parameters : Integers MAX-FLIPS, MAX-TRIES
Output    : A satisfying assignment for  $F$ , or FAIL
begin
  for  $i \leftarrow 1$  to MAX-TRIES do
     $\sigma \leftarrow$  a randomly generated truth assignment for  $F$ 
    for  $j \leftarrow 1$  to MAX-FLIPS do
      if  $\sigma$  satisfies  $F$  then return  $\sigma$  // success
       $v \leftarrow$  a variable flipping which results in the greatest decrease
        (possibly negative) in the number of unsatisfied clauses
      Flip  $v$  in  $\sigma$ 
    return FAIL // no satisfying assignment found
end

```

of unsatisfied clauses. In practice, this means that **GSAT** most often does not get stuck in local minima, although it may take a substantial amount of time on each plateau before moving on to the next one. This motivates studying various modifications in order to speed up this process [SK93, SKC94]. One of the most successful strategies is to introduce noise into the search in the form of uphill moves, which forms the basis of the now well-known local search method for SAT called **Walksat** [SKC93].

Walksat interleaves the greedy moves of **GSAT** with random walk moves of a standard Metropolis search. It further *focuses the search* by always selecting the variable to flip from an unsatisfied clause C (chosen at random). This seemingly simple idea of focusing the search turns out to be *crucial for scaling such techniques* to formulas beyond a few hundred variables. If there is a variable in C flipping which does not turn any currently satisfied clauses to unsatisfied, it flips this variable (a “freebie” move). Otherwise, with a certain probability, it flips a random literal of C (a “random walk” move), and with the remaining probability, it flips a variable in C that minimizes the *break-count*, i.e., the number of currently satisfied clauses that become unsatisfied (a “greedy” move). **Walksat** is presented in detail as Algorithm 6.2. One of its parameters, in addition to the maximum number of tries and flips, is the *noise* $p \in [0, 1]$, which controls how often are non-greedy moves considered during the stochastic search. It has been found empirically that for various instances from a single domain, a single value of p is optimal. For random 3-SAT formulas, the optimal noise is seen to be 0.57, and at this setting, **Walksat** is empirically observed to scale linearly for clause-to-variable ratios α up to (and even slightly beyond) 4.2 [SAO05], though not all the way up to the conjectured satisfiability threshold of nearly 4.26.²

The focusing strategy of **Walksat** based on selecting variables solely from unsatisfied clauses was inspired by the $O(n^2)$ randomized algorithm for 2-SAT by Papadimitriou [Pap91]. It can be readily shown that for any satisfiable formula

²Aurell et al. [AGK04] had observed earlier that **Walksat** scales linearly for random 3-SAT at least till clause-to-variable ratio 4.15.

Algorithm 6.2: Walksat (F)

```

Input      : A CNF formula  $F$ 
Parameters : Integers MAX-FLIPS, MAX-TRIES; noise parameter  $p \in [0, 1]$ 
Output     : A satisfying assignment for  $F$ , or FAIL
begin
  for  $i \leftarrow 1$  to MAX-TRIES do
     $\sigma \leftarrow$  a randomly generated truth assignment for  $F$ 
    for  $j \leftarrow 1$  to MAX-FLIPS do
      if  $\sigma$  satisfies  $F$  then return  $\sigma$  // success
       $C \leftarrow$  an unsatisfied clause of  $F$  chosen at random
      if  $\exists$  variable  $x \in C$  with break-count = 0 then
         $v \leftarrow x$  // freebie move
      else
        With probability  $p$ : // random walk move
           $v \leftarrow$  a variable in  $C$  chosen at random
        With probability  $1 - p$ : // greedy move
           $v \leftarrow$  a variable in  $C$  with the smallest break-count
        Flip  $v$  in  $\sigma$ 
    return FAIL // no satisfying assignment found
end

```

and starting from any truth assignment, there exists a sequence of flips using only variables from *unsatisfied* clauses such that one obtains a satisfying assignment. To see this, fix a particular satisfying assignment $\bar{\sigma}$. Let σ be any truth assignment. Every clause not satisfied by σ must contain a variable whose truth value is different in σ and $\bar{\sigma}$. Flipping such a variable in σ brings it one step closer to $\bar{\sigma}$. Repeating this at most n times makes σ identical to $\bar{\sigma}$, thereby turning σ into a satisfying assignment.

Knuth [Knu12] raised the question as to whether the rather innocuous looking *freebie rule* in *walksat* could make the search incomplete. (See also Hoos [Hoo98].) Cohen [Coh12] resolved this question by showing that one can construct a satisfiable set of clauses on which the freebie rule can result in a cycle through a set of unsatisfying assignments. This phenomenon requires a very careful construction and, in practice, such a cycle appears to be extremely rare.

Remark 6.1.1. We take a small detour to explain the elegant and insightful $O(n^2)$ time randomized local search algorithm for 2-SAT by Papadimitriou [Pap91]. The algorithm itself is very simple: while the current truth assignment does not satisfy all clauses, select an unsatisfied clause arbitrarily, select one of its variables uniformly at random, and flip this variable. Why does this take $O(n^2)$ flips in expectation before finding a solution? Assume without loss of generality that the all-zeros string (i.e., all variables set to FALSE) is a satisfying assignment for the formula at hand. Let $\bar{\sigma}$ denote this particular solution. Consider the Hamming distance $d(\sigma, \bar{\sigma})$ between the current truth assignment, σ , and the (unknown) solution $\bar{\sigma}$. Note that $d(\sigma, \bar{\sigma})$ equals the number of TRUE variables in σ . We claim that in each step of the algorithm, we reduce $d(\sigma, \bar{\sigma})$ by 1 with probability at least

a half, and increase it by one with probability at most a half. To see this, note that since $\bar{\sigma}$ is a solution, all clauses of the formula have at least one negative literal so that any unsatisfied clause selected by the algorithm must involve at least one variable that is currently set to TRUE, and selecting this variable to flip will result in decreasing $d(\sigma, \bar{\sigma})$ by 1. Given this claim, the algorithm is equivalent to a one-dimensional Markov chain of length $n + 1$ with the target node—the all-zeros string—on one extreme and the all-ones string at the other extreme, and that at every point we walk towards the target node with probability at least a half. It follows from standard Markov chain hitting time analysis that we will hit $\bar{\sigma}$ after $O(n^2)$ steps. (The algorithm could, of course, hit another solution before hitting $\bar{\sigma}$, which will reduce the runtime even further.)

When one compares the biased random walk strategy of `Walksat` on hard, random 3-CNF formulas against basic `GSAT`, the simulated annealing process of Kirkpatrick et al. [KGV83], and a pure random walk strategy, the biased random walk process significantly outperforms the other methods [SKC94]. In the years following the development of `Walksat`, many similar methods have been shown to be highly effective on not only random formulas but on several classes of structured instances, such as encodings of circuit design problems, Steiner tree problems, problems in finite algebra, and AI planning [cf. HS04].

6.2. Extensions of the Basic Local Search Method

Various extensions of the basic process discussed above have been explored, such as dynamic noise adaptation as in the solver `adapt-novelty` [Hoo02], incorporating unit clause elimination as in the solver `UnitWalk` [HK05], incorporating resolution-based reasoning [APSS05], and exploiting problem structure for increased efficiency [PTS07]. Recently, it was shown that the performance of stochastic solvers on many structured problems can be further enhanced by using new SAT encodings that are designed to be effective for local search [Pre07].

While adding random walk moves as discussed above turned out to be a successful method of guiding the search away from local basins of attraction and toward other parts of the search space, a different line of research considered techniques that relied on the idea of *clause re-weighting* as an extension of basic greedy search [CI96, Fra97, HTH02, ITA⁺06, Mor93, SLM92, TPBF04]. Here one assigns a positive weight to each clause and attempts to minimize the sum of the weights of the unsatisfied clauses. The clause weights are dynamically modified as the search progresses, increasing the weight of the clauses that are currently unsatisfied. (In some implementations, increasing the weight of a clause is done by simply adding identical copies of the clause.) In this way, if one waits sufficiently long, any unsatisfied clause gathers enough weight so as to sway the truth assignment in its favor. This is thought of as “flooding” the current local minimum by re-weighting unsatisfied clauses to create a new descent direction for local search. Variants of this approach differ in the re-weighting strategy used, e.g., how often and by how much the weights of unsatisfied clauses are increased, and how are all weights periodically decreased in order to prevent certain weights from becoming disproportionately high. The work on DLM or Discrete Lagrangian Method

grounded these techniques in a solid theoretical framework, whose details we defer to Section 6.3. The SDF or “smoothed descent and flood” system of Schuurmans and Southey [SS01] achieved significantly improved results by using multiplicative (rather than additive) re-weighting, by making local moves based on how strongly are the clauses currently satisfied in terms of the number of satisfied literals (rather than simply how many are satisfied), and by periodically shrinking all weights towards their common mean. Overall, two of the most well-known clause-reweighting schemes that have been proposed are SAPS (*scaling and probabilistic smoothing*, along with its *reactive* variant RSAPS) [HTH02] and PAWS (*pure additive weighting scheme*) [TPBF04].

Other approaches for improving the performance of GSAT were also explored in the early years. These include TSAT by Mazure et al. [MSG97], who maintain a tabu list in order to prevent GSAT from repeating earlier moves, and HSAT by Gent and Walsh [GW93], who consider breaking ties in favor of least recently flipped variables. These strategies provide improvement, but to a lesser extent than the basic random walk component added by Walksat.

In an attempt towards better understanding the pros and cons of many of these techniques, Schuurmans and Southey [SS01] proposed three simple, intuitive measures of the effectiveness of local search: *depth*, *mobility*, and *coverage*. (A) Depth measures how many clauses remain unsatisfied as the search proceeds. Typically, good local search strategies quickly descend to low depth and stay there for a long time; this corresponds to spending as much time as possible near the bottom of the search landscape. (B) Mobility measures how rapidly the process moves to new regions in the search space (while simultaneously trying to stay deep in the objective). Clearly, the larger the mobility, the better the chance that a local search strategy has of achieving success. (C) Coverage measures how systematically the process explores the entire space, in terms of the largest “gap”, i.e., the maximum Hamming distance between any unexplored assignment and the nearest evaluated assignment. Schuurmans and Southey [SS01] hypothesized that, in general, successful local search procedures work well not because they possess any special ability to predict whether a local basin of attraction contains a solution or not—rather they simply descend to promising regions and explore near the bottom of the objective as rapidly, broadly, and systematically as possible, until they stumble across a solution.

6.3. Discrete Lagrangian Methods

Shang and Wah [SW98] introduced a local search system for SAT based on the theory of Lagrange multipliers. They extended the standard Lagrange method, traditionally used for continuous optimization, to the discrete case of propositional satisfiability, in a system called DLM (Discrete Lagrangian Method). Although the final algorithm that comes out of this process can be viewed as a clause weighted version of local search as discussed in Section 6.2, this approach provided a theoretical foundation for many design choices that had appeared somewhat ad-hoc in the past. The change in the weights of clauses that are unsatisfied translates in this system to a change in the corresponding Lagrange multipliers, as one searches for a (local) optimum of the associated Lagrange function.

The basic idea behind DLM is the following. Consider an n -variable CNF formula F with clauses C_1, C_2, \dots, C_m . For $x \in \{0, 1\}^n$ and $i \in \{1, 2, \dots, m\}$, let $U_i(x)$ be a function that is 0 if C_i is satisfied by x , and 1 otherwise. Then the SAT problem for F can be written as the following optimization problem over $x \in \{0, 1\}^n$:

$$\begin{aligned} & \text{minimize} && N(x) = \sum_{i=1}^m U_i(x) && (6.1) \\ & \text{subject to} && U_i(x) = 0 && \forall i \in \{1, 2, \dots, m\} \end{aligned}$$

Notice that $N(x) \geq 0$ and equals 0 if and only if all clauses of F are satisfied. Thus, the objective function $N(x)$ is minimized if and only if x is a satisfying assignment for F . This formulation, somewhat strangely, also has each clause as an explicit constraint $U_i(x)$, so that any feasible solution is automatically also locally as well as globally optimal. This redundancy, however, is argued to be the strength of the system: the dynamic shift in emphasis between the objective and the constraints, depending on the relative values of the Lagrange multipliers, is the key feature of Lagrangian methods.

The theory of discrete Lagrange multipliers provides a recipe for converting the above constrained optimization problem into an unconstrained optimization problem, by introducing a Lagrange multiplier for each of the constraints and adding the constraints, appropriately multiplied, to the objective function. The resulting discrete Lagrangian function, which provides the new objective function to be optimized, is similar to what one would obtain in the traditional continuous optimization case:

$$L_d(x, \lambda) = N(x) + \sum_{i=1}^m \lambda_i U_i(x) \quad (6.2)$$

where $x \in \{0, 1\}^n$ are points in the variable space and $\lambda = (\lambda_1, \lambda_2, \dots, \lambda_m) \in \mathbb{R}^m$ is a vector of Lagrange multipliers, one for each constraint in (i.e., clause of) F . A point $(x^*, \lambda^*) \in \{0, 1\}^n \times \mathbb{R}^m$ is called a *saddle point* of the Lagrange function $L_d(x, \lambda)$ if it is a local minimum w.r.t. x^* and a local maximum w.r.t. λ^* . Formally, (x^*, λ^*) is a saddle point for L_d if

$$L_d(x^*, \lambda) \leq L_d(x^*, \lambda^*) \leq L_d(x, \lambda^*)$$

for all λ sufficiently close to λ^* and for all x that differ from x^* only in one dimension. It can be proven that x^* is a local minimum solution to the discrete constrained optimization formulation of SAT (6.1) if there exists a λ^* such that (x^*, λ^*) is a saddle point of the associated Lagrangian function $L_d(x, \lambda)$. Therefore, local search methods based on the Lagrangian system look for saddle points in the extended space of variables and Lagrange multipliers. By doing descents in the original variable space and ascents in the Lagrange-multiplier space, a saddle point equilibrium is reached when (locally) optimal solutions are found.

For SAT, this is done through a *difference gradient* $\Delta_x L_d(x, \lambda)$, defined to be a vector in $\{-1, 0, 1\}^n$ with the properties that it has at most one non-zero entry and $y = x \oplus \Delta_x L_d(x, \lambda)$ minimizes $L_d(y, \lambda)$ over all neighboring points y of x , including x itself. Here \oplus denotes vector addition; $x \oplus z = (x_1 + z_1, \dots, x_n + z_n)$.

The neighbors are “user-defined” and are usually taken to be all points that differ from x only in one dimension (i.e., are one variable flip away). Intuitively, the difference gradient $\Delta_x L_d(x, \lambda)$ “points in the direction” of the neighboring value in the variable space that minimizes the Lagrangian function for the current λ .

This yields an algorithmic approach for minimizing the discrete Lagrangian function $L_d(x, \lambda)$ associated with SAT (and hence minimizing the objective function $N(x)$ in the discrete constrained optimization formulation of SAT (6.1)). The algorithm proceeds iteratively in stages, updating $x \in \{0, 1\}^n$ and $\lambda \in \mathbb{R}^m$ in each stage using the difference gradient and the current status of each constraint in terms of whether or not it is satisfied, until a fixed point is found. Let $x(k)$ and $\lambda(k)$ denote the values of x and λ after the k^{th} iteration of the algorithm. Then the updates are as follows:

$$\begin{aligned}x(k+1) &= x(k) \oplus \Delta_x L_d(x(k), \lambda(k)) \\ \lambda(k+1) &= \lambda(k) + cU(x(k))\end{aligned}$$

where $c \in \mathbb{R}^+$ is a parameter that controls how fast the Lagrange multipliers are increased over iterations and U denotes the vector of the m constraint functions U_i defined earlier. The difference gradient $\Delta_x L_d$ determines which variable, if any, to flip in order to lower $L_d(x, \lambda)$ for the current λ . When a fixed point for these iterations is reached, i.e., when $x(k+1) = x(k)$ and $\lambda(k+1) = \lambda(k)$, it must be that all constraints U_i are satisfied. To see this, observe that if the i^{th} clause is unsatisfied after the k^{th} iteration, then $U_i(x(k)) = 1$, which implies $\lambda_i(k+1) = \lambda_i(k) + c$; thus, λ_i will keep increasing until U_i is satisfied. This provides *dynamic clause re-weighting* in this system, placing more and more emphasis on clauses that are unsatisfied until they become satisfied. Note that changing the Lagrange multipliers λ_i in turn affects x by changing the direction in which the difference gradient $\Delta_x L_d(x, \lambda)$ points, eventually leading to a point at which all constraints are satisfied. This is the essence of the DLM system for SAT.

The basic implementation of DLM [SW98] uses a simple controlled update rule for λ : increase λ_i by 1 for all unsatisfied clauses C_i after a pre-set number θ_1 of *up-hill* or *flat* moves (i.e., changes in x that do not decrease $L_d(x, \lambda)$) have been performed. In order to avoid letting some Lagrange multipliers become disproportionately large during the search, all λ_i 's are periodically decreased after a pre-set number θ_2 of increases in λ have been performed. Finally, the implementation uses *tabu lists* to store recently flipped variables, so as to avoid flipping them repeatedly.

Wu and Wah [WW99] observed that in many difficult to solve instances, such as from the **parity** and **hanoi** domains, basic DLM frequently gets stuck in *traps*, i.e., pairs (x, λ) such that there are one or more unsatisfied clauses but the associated L_d increases by changing x in any one dimension. They found that on these instances, some clauses are significantly more likely than others to be amongst the unsatisfied clauses in such a trap. (Note that this is different from counting how often is a clause unsatisfied; here we only consider the clause status inside a trap situation, ignoring how the search arrived at the trap.) Keeping track of such clauses and periodically performing a special increase in their associated Lagrange multipliers, guides the search away from traps and results in better performance.

[WW00] later generalized this strategy by recording not only information about traps but a history of all recently visited points in the variable space. Instead of performing a special increase periodically, this history information is now added directly as a *distance penalty* term to the Lagrangian function L_d . The penalty is larger for points that are in Hamming distance closer to the current value of x , thereby guiding the search away from recently visited points (in particular, points inside a trap that would be visited repeatedly).

6.4. The Phase Transition Phenomenon in Random k -SAT

One of the key motivations in the early 1990's for studying incomplete, stochastic methods for solving SAT instances was the observation that DPLL-based systematic solvers perform quite poorly on certain randomly generated formulas. For completeness, we provide a brief overview of these issues here; for a detailed discussion, refer to Part 1, Chapter 8 of this Handbook.

Consider a random k -CNF formula F on n variables generated by independently creating m clauses as follows: for each clause, select k distinct variables uniformly at random out of the n variables and negate each selected variable independently with probability $1/2$. When F is chosen from this distribution, Mitchell, Selman, and Levesque [MSL92] observed that the median hardness of the instances is very nicely characterized by a single parameter: the *clause-to-variable ratio*, m/n , typically denoted by α . They observed that instance hardness peaks in a critically constrained region determined by α alone. The top pane of Figure 6.1 depicts the now well-known “easy-hard-easy” pattern of SAT and other combinatorial problems, as the key parameter (in this case α) is varied. For random 3-SAT, this region has been experimentally shown to be around $\alpha \approx 4.26$ (for early results see Crawford and Auton [CA93] and Kirkpatrick and Selman [KS94]; new findings by Mertens et al. [MMZ06]), and has provided challenging benchmarks as a test-bed for SAT solvers. Cheeseman et al. [CKT91] observed a similar easy-hard-easy pattern in the random graph coloring problem. For random formulas, interestingly, a slight natural variant of the above “fixed-clause-length” model, called the variable-clause-length model, does *not* have a clear set of parameters that leads to a hard set of instances [FP83, Gol79, PB87]. This apparent difficulty in generating computationally hard instances for SAT solvers provided the impetus for much of the early work on local search methods for SAT. We refer the reader to Cook and Mitchell [CM97] for a detailed survey.

The critically constrained region marks a stark transition not only in the computational hardness of random SAT instances but also in their satisfiability itself. The bottom pane of Figure 6.1 shows the fraction of random formulas that are unsatisfiable, as a function of α . We see that nearly all formulas with α below the critical region (the under-constrained instances) are satisfiable. As α approaches and passes the critical region, there is a sudden change and nearly all formulas in this over-constrained region are unsatisfiable. Further, as n grows, this phase transition phenomenon becomes sharper and sharper, and coincides with the region in which the computational hardness peaks. The relative hardness of the instances in the unsatisfiable region to the right of the phase transition is consistent with the formal result of Chvátal and Szemerédi [CS88] who,

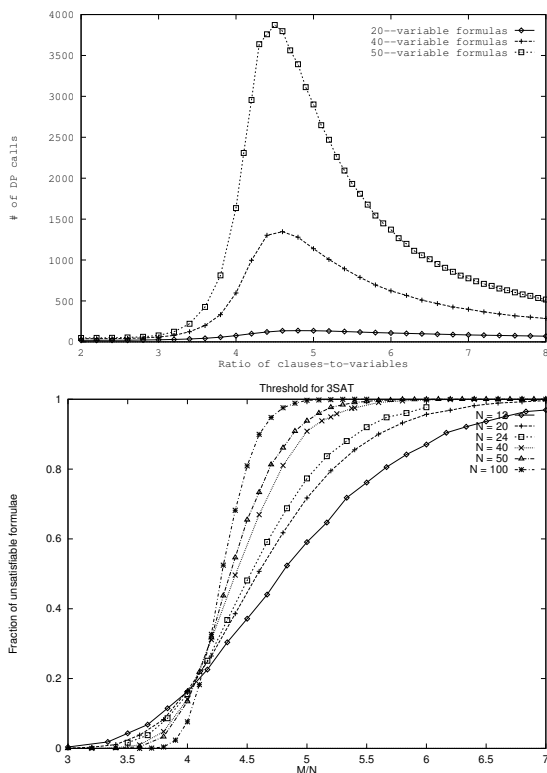


Figure 6.1. The phase transition phenomenon in random 3-SAT. Top: Computational hardness peaks at $\alpha \approx 4.26$. Bottom: Formulas change from being mostly satisfiable to mostly unsatisfiable. The transitions sharpen as the number of variables grows.

building upon the work of Haken [Hak85], proved that large unsatisfiable random k -CNF formulas almost surely require exponential size resolution refutations, and thus exponential length runs of any DPLL-based algorithm proving unsatisfiability. This formal result was subsequently refined and strengthened by others [cf. BKPS98, BP96, CEI96].

Relating the phase transition phenomenon for 3-SAT to statistical physics, Kirkpatrick and Selman [KS94] showed that the threshold has characteristics typical of phase transitions in the statistical mechanics of disordered materials (see also Monasson et al. [MZK⁺99] and Part 2, Chapter 18 of this Handbook). Physicists have studied phase transition phenomena in great detail because of the many interesting changes in a system's macroscopic behavior that occur at phase boundaries. One useful tool for the analysis of phase transition phenomena is called *finite-size scaling* analysis [SK96]. This approach is based on rescaling the horizontal axis by a factor that is a function of n . The function is such that the horizontal axis is stretched out for larger n . In effect, rescaling “slows down” the phase-transition for higher values of n , and thus gives us a better look inside

the transition. From the resulting universal curve, applying the scaling function backwards, the actual transition curve for each value of n can be obtained. In principle, this approach also localizes the 50%-satisfiable-point for any value of n , which allows one to generate very hard, random 3-SAT instances.

Interestingly, it is still not formally known whether there even exists a critical constant α_c such that as n grows, almost all 3-SAT formulas with $\alpha < \alpha_c$ are satisfiable and almost all 3-SAT formulas with $\alpha > \alpha_c$ are unsatisfiable. In this respect, Friedgut [Fri99] provided the first positive result, showing that there exists a *function* $\alpha_c(n)$ depending on n such that the above threshold property holds. (It is quite likely that the threshold in fact does not depend on n , and is a fixed constant.) In a series of papers, researchers have narrowed down the gap between upper bounds on the threshold for 3-SAT [e.g. BFU93, DBM03, FP83, JSV00, KKK96], the best so far being 4.51 [DBM03], and lower bounds [e.g. Ach00, AS00, BFU93, Fra83, FS96, HS03, KKL06], the best so far being 3.52 [HS03, KKL06]. On the other hand, for random 2-SAT, we do have a full rigorous understanding of the phase transition, which occurs at the clause-to-variable ratio of 1 [BBC⁺01, CR92]. Also, for general k , the threshold for random k -SAT is known to be in the range $2^k \ln 2 - O(k)$ [ANP05, AP04, GS05].

6.5. A New Technique for Random k -SAT: Survey Propagation

We end this chapter with a brief discussion of Survey Propagation (SP), an exciting new incomplete algorithm for solving hard combinatorial problems. The reader is referred to Part 2, Chapter 18 of this Handbook for a detailed treatment of work in this direction. Survey propagation was discovered in 2002 by Mézard, Parisi, and Zecchina [MPZ02], and is so far the only known method successful at solving random 3-SAT instances with one million variables and beyond in near-linear time in the most critically constrained region.³

The SP method is quite radical in that it tries to approximate, using an iterative process of local “message” updates, certain marginal probabilities related to the set of satisfying assignments. It then assigns values to variables with the most extreme probabilities, simplifies the formula, and repeats the process. This strategy is referred to as SP-inspired decimation. In effect, the algorithm behaves like the usual DPLL-based methods, which also assign variable values incrementally in an attempt to find a satisfying assignment. However, quite surprisingly, SP almost never has to backtrack. In other words, the “heuristic guidance” from SP is almost always correct. Note that, interestingly, computing marginals on satisfying assignments is strongly believed to be much harder than finding a single satisfying assignment (#P-complete vs. NP-complete). Nonetheless, SP is able to efficiently approximate certain marginals on random SAT instances and uses this information to successfully find a satisfying assignment.

SP was derived from rather complex statistical physics methods, specifically, the so-called *cavity method* developed for the study of spin glasses. The origin

³As mentioned earlier, it has been recently shown that by finely tuning its noise parameter, `Walksat` can also be made to scale well on hard, random 3-SAT instances with clause-to-variable ratios α slightly exceeding 4.2 [SAO05].

of SP in statistical physics and its remarkable and unparalleled success on extremely challenging random 3-SAT instances has sparked a lot of interest in the computer science research community, and has led to a series of papers in the last five years exploring various aspects of SP [e.g. ART06, AGK04, BZ04, KSS07, KSS08, KMRT⁺07, Man06, MS07, MMW07, MMZ05, MZ02, ZK07]. Many of these aspects still remain somewhat mysterious, making SP an active and promising research area for statistical physicists, theoretical computer scientists, and artificial intelligence practitioners alike.

While the method is still far from well-understood, close connections to belief propagation (BP) methods [Pea88] more familiar to computer scientists have been subsequently discovered. In particular, it was shown by Braunstein and Zecchina [BZ04] (later extended by Maneva, Mossel, and Wainwright [MMW07]) that SP equations are equivalent to BP equations for obtaining marginals over a special class of combinatorial objects called covers. In this respect, SP is the first successful example of the use of a probabilistic reasoning technique to solve a purely combinatorial search problem. The recent work of Kroc et al. [KSS07] empirically established that SP, despite the very loopy nature of random formulas which violate the standard tree-structure assumptions underlying the BP algorithm, is remarkably good at computing marginals over these covers objects on large random 3-SAT instances. Kroc et al. [KSS08] also demonstrated that information obtained from BP-style algorithms can be effectively used to enhance the performance of algorithms for the model counting problem, a generalization of the SAT problem where one is interested in counting the number of satisfying assignments.

Unfortunately, the success of SP is currently limited to random SAT instances. It is an exciting research challenge to further understand SP and apply it successfully to more structured, real-world problem instances.

6.6. Conclusion

Incomplete algorithms for satisfiability testing provide a complementary approach to complete methods, using an essentially disjoint set of techniques and being often well-suited to problem domains in which complete methods do not scale well. While a mixture of greedy descent and random walk provide the basis for most local search SAT solvers in use today, much work has gone into finding the right balance and in developing techniques to focus the search and efficiently bring it out of local minima and traps. Formalisms like the discrete Lagrangian method and ideas like clause weighting or flooding have played a crucial role in pushing the understanding and scalability of local search methods for SAT. An important role has also been played by the random k -SAT problem, particularly in providing hard benchmarks and a connection to the statistical physics community, leading to the survey propagation algorithm. Can we bring together ideas and techniques from systematic solvers and incomplete solvers to create a solver that has the best of both worlds? While some progress has been made in this direction, much remains to be done.

References

- [Ach00] D. Achlioptas. Setting 2 variables at a time yields a new lower bound for random 3-SAT. In *32nd STOC*, pages 28–37, Portland, OR, May 2000.
- [AGK04] E. Aurell, U. Gordon, and S. Kirkpatrick. Comparing beliefs, surveys, and random walks. In *18th NIPS*, Vancouver, BC, December 2004.
- [AJ90] H. M. Adorf and M. D. Johnston. A discrete stochastic neural network algorithm for constraint satisfaction problems. In *Intl. Joint Conf. on Neural Networks*, pages 917–924, San Diego, CA, 1990.
- [ANP05] D. Achlioptas, A. Naor, and Y. Peres. Rigorous location of phase transitions in hard optimization problems. *Nature*, 435:759–764, 2005.
- [AP04] D. Achlioptas and Y. Peres. The threshold for random k -SAT is $2^k(\ln 2 - O(k))$. *J. American Math. Soc.*, 17:947–973, 2004.
- [APSS05] Anbulagan, D. N. Pham, J. K. Slaney, and A. Sattar. Old resolution meets modern SLS. In *20th AAAI*, pages 354–359, Pittsburgh, PA, July 2005.
- [ART06] D. Achlioptas and F. Ricci-Tersenghi. On the solution-space geometry of random constraint satisfaction problems. In *38th STOC*, pages 130–139, Seattle, WA, May 2006.
- [AS00] D. Achlioptas and G. Sorkin. Optimal myopic algorithms for random 3-SAT. In *41st FOCS*, pages 590–600, Redondo Beach, CA, November 2000. IEEE.
- [BBC⁺01] B. Bollobás, C. Borgs, J. T. Chayes, J. H. Kim, and D. B. Wilson. The scaling window of the 2-SAT transition. *Random Struct. and Alg.*, 19(3-4):201–256, 2001.
- [BFU93] A. Broder, A. Frieze, and E. Upfal. On the satisfiability and maximum satisfiability of random 3-CNF formulas. In *Proc., 4th SODA*, January 1993.
- [BKPS98] P. Beame, R. Karp, T. Pitassi, and M. Saks. On the Complexity of Unsatisfiability Proofs for Random k -CNF Formulas. In *30th STOC*, pages 561–571, Dallas, TX, May 1998.
- [BP96] P. W. Beame and T. Pitassi. Simplified and improved resolution lower bounds. In *37th FOCS*, pages 274–282, Burlington, VT, October 1996. IEEE.
- [BZ04] A. Braunstein and R. Zecchina. Survey propagation as local equilibrium equations. *J. Stat. Mech.*, P06007, 2004.
- [CA93] J. M. Crawford and L. D. Auton. Experimental results on the crossover point in satisfiability problems. In *Proc. AAAI-93*, pages 21–27, Washington, DC, 1993.
- [CEI96] M. Clegg, J. Edmonds, and R. Impagliazzo. Using the Gröbner basis algorithm to find proofs of unsatisfiability. In *28th STOC*, pages 174–183, Philadelphia, PA, May 1996.
- [CI96] B. Cha and K. Iwama. Adding new clauses for faster local search. In *13th AAAI*, pages 332–337, Portland, OR, August 1996.

- [CKT91] P. Cheeseman, B. Kenefsky, and W. M. Taylor. Where the really hard problems are. In *Proceedings of IJCAI-91*, pages 331–337. Morgan Kaufmann, 1991.
- [CM97] S. Cook and D. Mitchell. Finding hard instances of the satisfiability problem: a survey. In *DIMACS Series in Discr. Math. and Theoretical Comp. Sci.*, volume 35, pages 1–17. American Math. Society, 1997.
- [Coh12] B. Cohen. Personal communication, 2012.
- [CR92] V. Chvátal and B. Reed. Mick gets some (the odds are on his side). In *33rd FOCS*, pages 620–627, Pittsburgh, PA, October 1992. IEEE.
- [CS88] V. Chvátal and E. Szemerédi. Many hard examples for resolution. *J. Assoc. Comput. Mach.*, 35(4):759–768, 1988.
- [DBM03] O. Dubois, Y. Boufkhad, and J. Mandler. Typical random 3-SAT formulae and the satisfiability threshold. Technical Report 7, ECCC, 2003.
- [DGH⁺02] E. Dantsin, A. Goerdt, E. A. Hirsch, R. Kannan, J. M. Kleinberg, C. H. Papadimitriou, P. Raghavan, and U. Schöning. A deterministic $(2 - 2/(k + 1))^n$ algorithm for k -SAT based on local search. *Theoretical Comput. Sci.*, 289(1):69–83, 2002.
- [DJW02] S. Droste, T. Jansen, and I. Wegener. On the analysis of the (1+1) evolutionary algorithm. *Theoretical Comput. Sci.*, 276(1-2):51–81, 2002.
- [ES03] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. springer, 2003.
- [FCS97] J. Frank, P. Cheeseman, and J. Stutz. Where gravity fails: Local search topology. *JAIR*, 7:249–281, 1997.
- [FP83] J. Franco and M. Paull. Probabilistic analysis of the Davis-Putnam procedure for solving the satisfiability problem. *Discr. Applied Mathematics*, 5:77–87, 1983.
- [Fra83] J. Franco. Probabilistic analysis of the pure literal heuristic for the satisfiability problem. *Annals of Operations Research*, 1:273–289, 1983.
- [Fra97] J. Frank. Learning short-term weights for GSAT. In *15th IJCAI*, pages 384–391, Nagoya, Japan, August 1997.
- [Fri99] E. Friedgut. Sharp thresholds of graph properties, and the k -sat problem. *Journal of the American Mathematical Society*, 12:1017–1054, 1999.
- [FS96] A. Frieze and S. Suen. Analysis of two simple heuristics on a random instance of k -SAT. *J. Alg.*, 20(2):312–355, 1996.
- [GB07] O.-C. Granmo and N. Bouhmala. Solving the satisfiability problem using finite learning automata. *Intl. J. Comp. Sci. App.*, 4(3):15–29, 2007.
- [GM94] M. L. Ginsberg and D. A. McAllester. GSAT and dynamic backtracking. In *4th KR*, pages 226–237, Bonn, Germany, May 1994.
- [GMR02] J. Gottlieb, E. Marchiori, and C. Rossi. Evolutionary algorithms for the satisfiability problem. *Evolutionary Computation*, 10(1):35–50, 2002.

- [Gol79] A. Goldberg. On the complexity of the satisfiability problem. Technical Report Report No. 16, Courant Computer Science, New York University, 1979.
- [GPFW97] J. Gu, P. W. Purdom, J. Franco, and B. W. Wah. Algorithms for the Satisfiability (SAT) Problem: A Survey. In *Satisfiability (SAT) Problem*, DIMACS, pages 19–151. American Mathematical Society, 1997.
- [GS05] C. P. Gomes and B. Selman. Can get satisfaction. *Nature*, 435:751–752, 2005.
- [Gu92] J. Gu. Efficient local search for very large-scale satisfiability problems. *Sigart Bulletin*, 3(1):8–12, 1992.
- [GW93] I. P. Gent and T. Walsh. Towards an understanding of hill-climbing procedures for SAT. In *11th AAAI*, pages 28–33, Washington, DC, July 1993.
- [Hak85] A. Haken. The intractability of resolution. *Theoretical Comput. Sci.*, 39:297–305, 1985.
- [HK05] E. A. Hirsch and A. Kojevnikov. UnitWalk: A new SAT solver that uses local search guided by unit clause elimination. *Annals Math. and AI*, 43(1):91–111, 2005.
- [HLDV02] D. Habet, C. M. Li, L. Devendeville, and M. Vasquez. A hybrid approach for SAT. In *8th CP*, volume 2470 of *LNCS*, pages 172–184, Ithaca, NY, September 2002.
- [Hoo88] J. N. Hooker. A quantitative approach to logical inference. *Decision Support Systems*, 4:45–69, 1988.
- [Hoo98] H. Hoos. *Stochastic Local Search – Methods, Models, Applications*. PhD thesis, Darmstadt University of Technology, Germany, 1998.
- [Hoo99] H. H. Hoos. On the run-time behaviour of stochastic local search algorithms for SAT. In *Proceedings of AAAI-99*, pages 661–666. AAAI Press, 1999.
- [Hoo02] H. H. Hoos. An adaptive noise mechanism for WalkSAT. In *18th AAAI*, pages 655–660, Edmonton, Canada, July 2002.
- [HS03] M. Hajiaghayi and G. B. Sorkin. The satisfiability threshold of random 3-SAT is at least 3.52. Technical Report RC22942, IBM Research Report, 2003. <http://arxiv.org/pdf/math.CO/0310193>.
- [HS04] H. H. Hoos and T. Stützle. *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann, San Francisco, CA, 2004.
- [HTH02] F. Hutter, D. A. D. Tompkins, and H. H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In *8th CP*, volume 2470 of *LNCS*, pages 233–248, Ithaca, NY, September 2002.
- [ITA⁺06] A. Ishtaiwi, J. Thornton, Anbulagan, A. Sattar, and D. N. Pham. Adaptive clause weight redistribution. In *12th CP*, volume 4204 of *LNCS*, pages 229–243, Nantes, France, September 2006.
- [JAMS91] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing: an experimental evaluation; part II. *Operations Research*, 39, 1991.
- [JSV00] S. Janson, Y. C. Stamatiou, and M. Vamvakari. Bounding the un-

satisfiability threshold of random 3-SAT. *Random Struct. and Alg.*, 17(2):103–116, 2000.

- [JT93] D. S. Johnson and M. A. Trick, editors. *Cliques, Coloring and Satisfiability: the Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Series in DMTCS*. Amer. Math. Soc., 1993. DIMACS workshop in 1993, published in 1996.
- [KGV83] S. Kirkpatrick, D. Gelatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.
- [KKK96] L. M. Kirousis, E. Kranakis, and D. Krizanc. Approximating the unsatisfiability threshold of random formulas. In *Proceedings of the Fourth Annual European Symposium on Algorithms*, pages 27–38, Barcelona, Spain, September 1996.
- [KKL06] A. C. Kaporis, L. M. Kirousis, and E. G. Lalas. The probabilistic analysis of a greedy satisfiability algorithm. *Random Structures and Algorithms*, 28(4):444–480, 2006.
- [KKRR90] A. Kamath, N. Karmarkar, K. Ramakrishnan, and M. Resende. Computational experience with an interior point algorithm on the satisfiability problem. In *Proceedings of Integer Programming and Combinatorial Optimization*, pages 333–349, Waterloo, Canada, 1990. Mathematical Programming Society.
- [KMRT⁺07] F. Krzakala, A. Montanari, F. Ricci-Tersenghi, G. Semerjian, and L. Zdeborova. Gibbs states and the set of solutions of random constraint satisfaction problems. *PNAS*, 104(25):10318–10323, June 2007.
- [Knu12] D. Knuth. Personal communication, 2012.
- [Kon94] K. Konolige. Easy to be hard: Difficult problems for greedy algorithms. In *4th KR*, pages 374–378, Bonn, Germany, May 1994.
- [KS94] S. Kirkpatrick and B. Selman. Critical behavior in the satisfiability of random Boolean expressions. *Science*, 264:1297–1301, May 1994. Also p. 1249: "Math. Logic: Pinning Down a Treacherous Border in Logical Statements" by B. Cipra.
- [KSS07] L. Kroc, A. Sabharwal, and B. Selman. Survey propagation revisited. In *23rd UAI*, pages 217–226, Vancouver, BC, July 2007.
- [KSS08] L. Kroc, A. Sabharwal, and B. Selman. Leveraging belief propagation, backtrack search, and statistics for model counting. In *5th CPAIOR*, volume 5015 of *LNCS*, pages 127–141, Paris, France, May 2008.
- [LK73] S. Lin and B. W. Kernighan. An efficient heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21:498–516, 1973.
- [LSB03] X. Y. Li, M. F. M. Stallmann, and F. Brglez. QingTing: A local search SAT solver using an effective switching strategy and an efficient unit propagation. In *6th SAT*, pages 53–68, Santa Margherita, Italy, May 2003.
- [LSH06] F. Lardeux, F. Saubion, and J.-K. Hao. GASAT: A genetic local search algorithm for the satisfiability problem. *Evolutionary Computation*, 14(2):223–253, 2006.

- [LWZ07] C. M. Li, W. Wei, and H. Zhang. Combining adaptive noise and look-ahead in local search for SAT. In *10th SAT*, volume 4501 of *LNCS*, pages 121–133, Lisbon, Portugal, May 2007.
- [Man06] E. Maneva. *Belief Propagation Algorithms for Constraint Satisfaction Problems*. PhD thesis, University of California, Berkeley, 2006.
- [MJPL90] S. Minton, M. D. Johnston, A. B. Philips, and P. Laird. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings AAAI-90*, pages 17–24. AAAI Press, 1990.
- [MMW07] E. Maneva, E. Mossel, and M. J. Wainwright. A new look at survey propagation and its generalizations. *J. Assoc. Comput. Mach.*, 54(4):17, July 2007.
- [MMZ05] M. Mézard, T. Mora, and R. Zecchina. Clustering of solutions in the random satisfiability problem. *Phys. Rev. Lett.*, 94:197205, May 2005.
- [MMZ06] S. Mertens, M. Mézard, and R. Zecchina. Threshold values of random K-SAT from the cavity method. *Random Struct. and Alg.*, 28(3):340–373, 2006.
- [Mor93] P. Morris. The breakout method for escaping from local minima. In *11th AAAI*, pages 428–433, Washington, DC, July 1993.
- [MPZ02] M. Mézard, G. Parisi, and R. Zecchina. Analytic and algorithmic solution of random satisfiability problems. *Science*, 297(5582):812–815, 2002.
- [MS07] E. Maneva and A. Sinclair. On the satisfiability threshold and survey propagation for random 3-SAT, 2007. <http://arxiv.org/abs/cs.CC/0609072>.
- [MSG96] B. Mazure, L. Sais, and É. Grégoire. Boosting complete techniques thanks to local search methods. In *Proc. Math and AI*, 1996.
- [MSG97] B. Mazure, L. Sais, and É. Grégoire. Tabu search for SAT. In *14th AAAI*, pages 281–285, Providence, RI, July 1997.
- [MSK97] D. A. McAllester, B. Selman, and H. A. Kautz. Evidence for invariants in local search. In *AAAI/IAAI*, pages 321–326, Providence, RI, July 1997.
- [MSL92] D. Mitchell, B. Selman, and H. J. Levesque. Hard and easy distributions of SAT problems. In *Proc. AAAI-92*, pages 459–465, San Jose, CA, 1992.
- [MZ02] M. Mézard and R. Zecchina. Random k-satisfiability problem: From an analytic solution to an efficient algorithm. *Phys. Rev. E*, 66:056126, November 2002.
- [MZK⁺99] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. Determining computational complexity from characteristic phase transitions. *Nature*, 400(8):133–137, 1999.
- [Pap91] C. H. Papadimitriou. On selecting a satisfying truth assignment. In *32nd FOCS*, pages 163–169, San Juan, Puerto Rico, October 1991. IEEE.
- [PB87] P. W. Purdom Jr. and C. A. Brown. Polynomial average-time satisfiability problems. *Information Science*, 41:23–42, 1987.

- [Pea88] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.
- [Pre01] S. D. Prestwich. Local search and backtracking vs non-systematic backtracking. In *AAAI 2001 Fall Symp. on Using Uncertainty within Computation*, 2001.
- [Pre07] S. D. Prestwich. Variable dependency in local search: Prevention is better than cure. In *10th SAT*, Lisbon, Portugal, May 2007.
- [PS82] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization*. Prentice-Hall, Inc., 1982.
- [PTS07] D. N. Pham, J. Thornton, and A. Sattar. Building structure into local search for SAT. In *20th IJCAI*, pages 2359–2364, Hyderabad, India, January 2007.
- [PW96] A. J. Parkes and J. P. Walser. Tuning local search for satisfiability testing. In *13th AAAI*, pages 356–362, Portland, OR, August 1996.
- [RD96] I. Rish and R. Dechter. To guess or to think? hybrid algorithms for SAT. In *Proc. Conference on Principles of Constraint Programming (CP-96)*, pages 555–556, 1996.
- [RF93] M. G. C. Resende and T. A. Feo. A GRASP for satisfiability. In Johnson and Trick [JT93], pages 499–520. DIMACS workshop in 1993, published in 1996.
- [SAO05] S. Seitz, M. Alava, and P. Orponen. Focused local search for random 3-satisfiability. *J. Stat. Mech.*, P06006, 2005.
- [Sch99] U. Schöning. A probabilistic algorithm for k -SAT and constraint satisfaction problems. In *40th FOCS*, pages 410–414, New York, NY, October 1999. IEEE.
- [Sch02] U. Schöning. A probabilistic algorithm for k -SAT based on limited local search and restart. *Algorithmica*, 32(4):615–623, 2002.
- [SK93] B. Selman and H. A. Kautz. Domain-independent extensions to GSAT: Solving large structured satisfiability problems. In *13th IJCAI*, pages 290–295, France, 1993.
- [SK96] B. Selman and S. Kirkpatrick. Critical behavior in the computational cost of satisfiability testing. *AI J.*, 81:273–295, 1996.
- [SKC93] B. Selman, H. A. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In Johnson and Trick [JT93], pages 521–532. DIMACS workshop in 1993, published in 1996.
- [SKC94] B. Selman, H. A. Kautz, and B. Cohen. Noise strategies for local search. In *Proc. AAAI-94*, pages 337–343, Seattle, WA, 1994.
- [SLM92] B. Selman, H. J. Levesque, and D. G. Mitchell. A new method for solving hard satisfiability problems. In *10th AAAI*, pages 440–446, San Jose, CA, July 1992.
- [Spe93] W. M. Spears. Simulated annealing for hard satisfiability problems. In Johnson and Trick [JT93], pages 533–558. DIMACS workshop in 1993, published in 1996.
- [SS01] D. Schuurmans and F. Southey. Local search characteristics of incomplete SAT procedures. *AI J.*, 132(2):121–150, 2001.
- [SSH01] D. Schuurmans, F. Southey, and R. C. Holte. The exponentiated subgradient algorithm for heuristic boolean programming. In *17th*

- IJCAI*, pages 334–341, Seattle, WA, August 2001.
- [SW98] Y. Shang and B. W. Wah. A discrete Lagrangian-based global-search method for solving satisfiability problems. *J. of Global Optimization*, 12(1):61–99, 1998.
- [TPBF04] J. Thornton, D. N. Pham, S. Bain, and V. Ferreira Jr. Additive versus multiplicative clause weighting for SAT. In *19th AAAI*, pages 191–196, San Jose, CA, July 2004.
- [WW99] Z. Wu and B. W. Wah. Trap escaping strategies in discrete Lagrangian methods for solving hard satisfiability and maximum satisfiability problems. In *16th AAAI*, pages 673–678, Orlando, FL, July 1999.
- [WW00] Z. Wu and B. W. Wah. An efficient global-search strategy in discrete Lagrangian methods for solving hard satisfiability problems. In *17th AAAI*, pages 310–315, Austin, TX, July 2000.
- [ZK07] L. Zdeborova and F. Krzakala. Phase transition in the coloring of random graphs. *Phy. Rev. E*, 76:031131, September 2007.

Chapter 7

Proof Complexity and SAT Solving

Sam Buss and Jakob Nordström

7.1. Introduction

The satisfiability problem (SAT) — i.e., to determine whether a given formula in propositional logic has a satisfying assignment or not — is of central importance to both the theory of computer science and the practice of automatic theorem proving and proof search. Proof complexity — i.e., the study of the complexity of proofs and the difficulty of searching for proofs — joins the theoretical and practical aspects of satisfiability.

For theoretical computer science, SAT is the canonical NP-complete problem, even for conjunctive normal form (CNF) formulas [Coo71, Lev73]. In fact, SAT is very efficient at expressing problems in NP in that many of the standard NP-complete problems, including the question of whether a (nondeterministic) Turing machine halts within n steps, have very efficient, almost linear time, reductions to the satisfiability of CNF formulas.¹ A popular hypothesis in the computational complexity community is the *Strong Exponential Time Hypothesis (SETH)*, which says that any algorithm for solving CNF SAT must have worst-case running time (roughly) 2^n on instances with n variables [IP01, CIP09]. This hypothesis has been widely studied in recent years, and has served as a basis for proving conditional hardness results for many other problems. In other words, CNF SAT serves as the canonical hard decision problem, and is frequently conjectured to require exponential time to solve.

In contrast, for practical theorem proving, CNF SAT is the core method for encoding and solving problems. On one hand, the expressiveness of CNF formulas means that a large variety of problems can be faithfully and straightforwardly translated into CNF SAT problems. On the other hand, the message that SAT is supposed to be hard to solve does not seem to have reached practitioners of SAT solving; instead, there has been enormous improvements in performance in SAT algorithms over the last decades. Amazingly, state-of-the-art algorithms for

¹Formally, these reductions run in *quasilinear* time, i.e. time at most $n(\log n)^k$ for some constant k . For these quasilinear time reductions of the Turing machine halting problem to CNF SAT, see [Sch78, PF79, Rob79, Rob91, Coo88].

deciding satisfiability — so-called *SAT solvers* — can routinely handle real-world instances involving hundreds of thousands or even millions of variables. It is a dramatic development that SAT solvers can often run in (close to) linear time!

Thus, theoreticians view SAT as being infeasible, while practitioners view it as being (often) feasible. There is no contradiction here. First, it is possible to construct tiny formulas with just a few hundred variables that are totally beyond reach for even the best of today’s solvers. Conversely, the large instances which are solved by SAT solvers are based on problems that seem to be “easy” in some sense, although they are very large. However, we currently lack a good theoretical explanation for what makes these problems “easy”. Most SAT solvers are general-purpose and written in a very generic way that does not seem to exploit special properties of the underlying problem. Nonetheless, although SAT solvers will sometimes fail miserably, they succeed much more frequently than might be expected. This raises the questions of how practical SAT solvers can perform so well on many large problems and of what distinguishes problems that can be solved by SAT solvers from problems that cannot.

The best current SAT solvers are based on *conflict-driven clause learning (CDCL)* [MSS99, MMZ⁺01].² Some solvers also incorporate elements of algebraic reasoning (e.g., Gaussian elimination) and/or geometric reasoning (e.g., linear inequalities), or use algebraic or geometric methods as the foundation rather than CDCL. Another augmentation of CDCL that has attracted much interest is *extended resolution (ER)*. How can we analyze the power of such algorithms? Our best approach is to study the underlying methods of reasoning and what they are able or unable to do in principle. This leads to the study of proof systems such as resolution, extended resolution, Nullstellensatz, polynomial calculus, cutting planes, et cetera. *Proof complexity*, as initiated in modern form by [CR79, Rec75], studies these systems mostly from the viewpoint of the complexity of static, completed proofs. With a few exceptions (perhaps most notably *automatability*³ as defined in [BPR00]), research in proof complexity ignores the constructive, algorithmic aspects of SAT solvers. Nonetheless, proof complexity has turned out to be a very useful tool for studying practical SAT solvers, in particular, because it is a way to obtain mathematically rigorous bounds on solver performance. Lower bounds on the complexity of proofs in a proof system show fundamental limitations on what one can hope to achieve using SAT solvers based on the corresponding method of reasoning. Conversely, upper bounds can be viewed as indications of what should be possible to achieve using a method of reasoning, if only proof search using this method could be implemented efficiently enough.

We want to stress, though, that the tight connections to proof complexity come with a price. Since proof complexity studies what reasoning can be achieved by a method in principle, ignoring the algorithmic challenge of actually implementing such reasoning, it becomes very hard to say anything meaningful about satisfiable formulas. In principle, it is very hard to rule out that a solver could simply zoom in on a satisfying assignment right away, and this will of course only take linear time. For unsatisfiable formulas, however, a (complete) solver

²A similar technique for constraint satisfaction problems (CSPs) was independently developed in [BS97].

³Automatability is sometimes also called *automatizability*.

will have to certify that there is no satisfying assignment, and for many SAT solving methods used in practice this is a computational task that is amenable to mathematical analysis. This article, therefore, will focus almost exclusively on unsatisfiable formulas. This is, admittedly, a severe limitation, but it is dictated by the limitations of our current mathematical knowledge. There has been some work on proof complexity of satisfiable formulas by reducing this to problems about unsatisfiable subformulas (e.g., in [AHI05]), but the literature here is very limited. Having said this, we want to point out that the setting of unsatisfiable formulas is nevertheless very interesting, since in many real-world applications proving unsatisfiability is the objective, and conventional wisdom is that such instances are often the hardest ones.

This chapter is intended as an overview of the connection between SAT solving and proof complexity aimed at readers who wish to become more familiar with either (or both) of these areas. We focus on the proof systems underlying current approaches to SAT solving. Our goal is first to explain how SAT solvers correspond to proof systems and second to review some of the complexity results known for these proof systems. We will discuss *resolution* (corresponding to basic CDCL proof search), *Nullstellensatz* and *polynomial calculus* (corresponding to algebraic approaches such as *Gröbner basis* computations), *cutting planes* (corresponding to *pseudo-Boolean solving*), *extended resolution* (corresponding to the *DRAT* proof logging system used for CDCL solvers with pre-/inprocessing), and will also briefly touch on *Frege systems* and *bounded-depth Frege systems*.

We want to emphasize that we do not have space to cover more than a small part of the research being done in proof complexity. Some useful material for further reading are the survey articles [BP98a, Seg07] and the recent book [Kra19]. Additionally, we would like to mention the authors' own surveys [Bus99, Bus12, Nor13]. The present chapter is adapted from and partially overlaps with the second author's survey [Nor15], but has been thoroughly rewritten and substantially expanded with new material.

7.1.1. Outline of This Survey Chapter

The rest of this chapter is organized as follows. Section 7.2 presents a quick review of preliminaries. We discuss the resolution proof system and describe the connection to CDCL SAT solvers in Section 7.3, and then give an overview of some of the proof complexity results known for resolution in Section 7.4. In Section 7.5 we consider the algebraic proof systems Nullstellensatz and polynomial calculus, and also briefly touch on algebraic SAT solving. In Section 7.6 we move on to the geometric proof system cutting planes and the connections to conflict-driven pseudo-Boolean solving, after which we give an overview of what is known in proof complexity about different flavours of the cutting planes proof system in Section 7.7. We review extended resolution and DRAT in Section 7.8, and then continue to Frege and extended Frege proof systems and bounded-depth Frege systems in Sections 7.9 and 7.10. Section 7.11 gives some concluding remarks.

7.2. Preliminaries

We use the notation $[n] = \{1, 2, \dots, n\}$ for n a positive integer. We write $\mathbb{N} = \{0, 1, 2, 3, \dots\}$ to denote the set of all natural numbers and $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$ for the set of positive integers.

7.2.1. Propositional Logic

A *Boolean variable* x ranges over values *true* and *false*; unless otherwise stated, we identify 1 with true and 0 with false. A *literal* a over a Boolean variable x is either the variable x itself (a *positive literal*) or its negation \bar{x} (a *negative literal*). We define $\bar{\bar{x}} = x$. It will sometimes be convenient to use the alternative notation x^σ , $\sigma \in \{0, 1\}$, for literals, where $x^1 = x$ and $x^0 = \bar{x}$. (In other words, x^σ is the literal that evaluates to true under the assignment $x = \sigma$.)

A *clause* $C = a_1 \vee \dots \vee a_k$ is a disjunction of literals over distinct variables. The empty clause, with $k = 0$, is denoted \perp . By convention, clauses are not *tautological*; i.e., do not contain any variable and its negation. A clause C' *subsumes* another clause C if every literal from C' also appears in C . In this case, C' is at least as strong as clause C . A *k-clause* is a clause that contains at most k literals. A *CNF formula* $F = C_1 \wedge \dots \wedge C_m$ is a conjunction of clauses. F is a *k-CNF formula* if it consists of k -clauses. We think of clauses and CNF formulas as sets: the order of elements is irrelevant and there are no repetitions.⁴

The *width* of a clause is the number of literals in it. Thus, a *k-clause* has width at most k . In the rest of this chapter, k is assumed to be some arbitrary but fixed constant unless stated otherwise. There is a standard way to turn any CNF formula F into 3-CNF by converting every clause

$$a_1 \vee a_2 \vee \dots \vee a_w \tag{7.1a}$$

of width $w > 3$ into the set of $w-2$ many 3-clauses

$$\{a_1 \vee a_2 \vee y_2\} \cup \{\bar{y}_{j-1} \vee a_j \vee y_j \mid 3 \leq j < w-1\} \cup \{\bar{y}_{w-2} \vee a_{w-1} \vee a_w\}, \tag{7.1b}$$

where the y_j 's denote new variables. This conversion to 3-CNF often does not change much from a theoretical point of view (however, there are some exceptions to this rule, which we will discuss in the relevant context).

We typically use N to denote the *size* of a formula F , namely the total number of occurrences of literals in F . (For a k -CNF formula with $k = O(1)$, N may instead denote the number of clauses in F ; this differs from the size of F by at most a constant factor.)

A *truth assignment* is any mapping from variables to truth values 0 or 1. We allow truth assignments to be *partial*, i.e., with some variables left unassigned. A truth assignment is *total* if it assigns values to all variables under consideration. We represent a (partial) truth assignment ρ as the set of literals set to true by ρ . We write $\rho(x^\sigma) = 1$ if $x^\sigma \in \rho$, and $\rho(x^\sigma) = 0$ if $x^{1-\sigma} \in \rho$. If ρ does not assign any truth value to x , we write $\rho(x^\sigma) = *$. The assignment ρ *satisfies* a clause C

⁴It can be noted, though, that some SAT solvers implement formulas as *multisets* of clauses.

provided it sets at least one literal of C true; it *falsifies* C provided it sets every literal in C false. It *satisfies* a formula F provided it satisfies every clause in F .

A formula F is *satisfiable* provided some assignment satisfies it; otherwise it is *unsatisfiable*. We say that a formula F *logically implies* a clause C if every total truth assignment ρ which satisfies F also satisfies C . In this case, we write $F \models C$. Since \perp is unsatisfiable, $F \models \perp$ is equivalent to F being unsatisfiable.

7.2.2. Proof Systems

The principal task of a SAT solver is to determine whether a given formula F is satisfiable or unsatisfiable. When F is satisfiable, there is always a short proof of its satisfiability; namely, a satisfying truth assignment. Establishing that F is unsatisfiable is generally done using some kind of proof system; *proof complexity* studies the complexity of the possible proofs in different proof systems.

In what follows, we give a high-level mathematical description of what a formal proof system is. Later sections will focus in more detail on concrete proof systems that are relevant in the context of SAT solving, and these sections can be read independently without appreciating all of the finer details below.

In the most general setting of proof complexity, one focuses on some *language* L , i.e., an (infinite) set of strings over some (finite) alphabet of symbols, and studies the complexity of proving that a given string x is in the set L . In this abstract setting, a *proof system for* L is a binary predicate $\mathcal{P}(x, \pi)$ that takes as input two strings x and π , is computable (deterministically) in time polynomial in the sizes $|x|$ and $|\pi|$ of the inputs, and has two key properties:

Completeness: for all $x \in L$ there is a string π (a *proof*) for which $\mathcal{P}(x, \pi)$ evaluates to true;

Soundness: for all $x \notin L$ it holds for all strings π that $\mathcal{P}(x, \pi)$ evaluates to false.

Intuitively, completeness means that all valid statements are provable, whereas soundness means that no fake proofs of invalid claims are accepted.

Informally, the strength of a proof system is measured by how small proofs it can produce. The *size* of a formula or proof is equal to the number of symbols it contains. The ideal scenario would be to have a proof system where all theorems can be proven with at most a polynomial blow-up in size of the proof compared to the size of the input. Formally, a proof system is said to be *polynomially bounded* if for every $x \in L$ there is a proof π_x for x that has size at most polynomial in $|x|$.

It is common to measure the relative strengths of proof systems in terms of the lengths of their proofs. Suppose \mathcal{P} and \mathcal{P}' are proof systems. We say that \mathcal{P}' *polynomially simulates* \mathcal{P} if, for all $x \in L$, the size of the shortest \mathcal{P}' -proof of x is polynomially bounded by the size of the shortest \mathcal{P} -proof of x .⁵

In the context of this survey, we will mostly focus on proving unsatisfiability of CNF formulas. This means that we will deal with proof systems for the formal language L consisting of all unsatisfiable CNF formulas. In this context, a proof

⁵In the literature, the terms “simulate”, “p-simulate” and “polynomially simulate” are used more or less interchangeably, but sometimes coupled with the proviso that there is a polynomial time algorithm which can produce a \mathcal{P}' -proof given a \mathcal{P} -proof as input.

system is a polynomial-time algorithm for verifying the unsatisfiability of a given formula F with the aid of a given proof π . Most of the proof systems relevant for SAT solvers (such as resolution, Nullstellensatz, polynomial calculus, cutting planes, et cetera) are such *refutation systems* designed for refuting unsatisfiable formulas F . However, some proof systems in propositional logic (such as Frege systems) instead produce proofs of valid formulas. For such proof system, the language L under study consists of all tautological formulas. There is a duality between refutations and proofs in that the CNF formula F is unsatisfiable if and only if its negation, the DNF formula $\neg F$, is valid (i.e., is a tautology). We will use the term *proof system* to refer to both kinds of systems above. When discussing refutation systems, we will frequently use the term *proof (of unsatisfiability) for F* to refer to a refutation of F .

7.2.3. Complexity and Asymptotic Notation

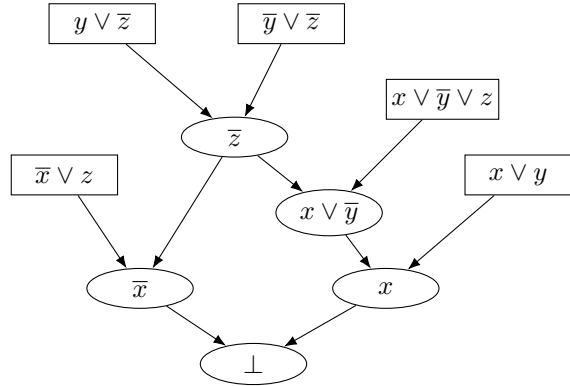
One of the central tasks of proof complexity is to prove upper or lower bounds on the complexity of proofs. That is, for a fixed proof system \mathcal{P} and a class \mathcal{F} of unsatisfiable formulas, we may let $f(N)$ denote the worst-case size that might be needed for a \mathcal{P} -refutation of any formula $F \in \mathcal{F}$ of size N ; then we wish to provide sharp bounds on the growth rate of $f(N)$. Small upper bounds on $f(N)$ indicate that \mathcal{P} is an efficient proof system for \mathcal{F} -formulas. Small upper bounds also suggest the possibility of efficient search procedures for finding refutations of formulas $F \in \mathcal{F}$. Large lower bounds on $f(N)$ indicate that \mathcal{P} cannot be efficient for all \mathcal{F} -formulas.

Proof complexity often deals with asymptotic upper and lower bounds, generally expressed by using “big-O” notation. The commonly used notations include $f(n) = O(g(n))$, $f(n) = o(g(n))$, $f(n) = \Omega(g(n))$, $f(n) = \omega(g(n))$, and $f(n) = \Theta(g(n))$, where $f(n)$ and $g(n)$ are always nonnegative. We write $f(n) = O(g(n))$ to express that $\exists c > 0 \exists n_0 \forall n \geq n_0 f(n) \leq c \cdot g(n)$, i.e., that f grows at most as quickly as g asymptotically. We write $f(n) = o(g(n))$ to express that $\forall \epsilon > 0 \exists n_0 \forall n \geq n_0 f(n) \leq \epsilon \cdot g(n)$, i.e., that f grows strictly more slowly than g . When $g(n) > 0$ always holds, $f(n) = o(g(n))$ is equivalent to $\lim_{n \rightarrow \infty} f(n)/g(n) = 0$. The notations $f(n) = \Omega(g(n))$ and $f(n) = \omega(g(n))$, mean that $g(n) = O(f(n))$ and $g(n) = o(f(n))$, respectively, or, in words, that f grows at least as fast or strictly faster than g . We write $f(n) = \Theta(g(n))$ to denote that both $f(n) = O(g(n))$ and $g(n) = O(f(n))$ hold, i.e., that asymptotically speaking f and g are essentially the same function up to a constant factor. Big-O notation is often used in subexpressions. For example, we write $f(n) = 2^{(1-o(1))n}$ to mean that for any fixed $\epsilon > 0$, we have $f(n) > 2^{(1-\epsilon)n}$ for all sufficiently large n .

7.3. Resolution and CDCL SAT solvers

We start our survey by discussing the *resolution proof system* [Bla37, DP60, DLL62, Rob65], which is the most important proof system from the point of view of SAT solving. Resolution is a refutation system that works directly with clauses. A resolution derivation π of a clause D from a CNF formula F is a sequence of clauses $\pi = (D_1, D_2, \dots, D_{L-1}, D_L)$ such that $D = D_L$ and each clause D_i is either

1. $x \vee y$ Axiom
2. $x \vee \bar{y} \vee z$ Axiom
3. $\bar{x} \vee z$ Axiom
4. $y \vee \bar{z}$ Axiom
5. $\bar{y} \vee \bar{z}$ Axiom
6. \bar{z} Res(4, 5)
7. \bar{x} Res(3, 6)
8. $x \vee \bar{y}$ Res(2, 6)
9. x Res(1, 8)
10. \perp Res(7, 9)



(a) Refutation as annotated list.

(b) Refutation as directed acyclic graph (DAG).

Figure 7.1. Resolution refutation of the CNF formula (7.3).

1. an *axiom clause* $D_i \in F$, or
2. a clause of the form $D_i = B \vee C$ derived from clauses $D_j = B \vee x$ and $D_k = C \vee \bar{x}$ for $j, k < i$ by the *resolution rule*

$$\frac{B \vee x \quad C \vee \bar{x}}{B \vee C} . \tag{7.2}$$

We call $B \vee C$ the *resolvent over x* of $B \vee x$ and $C \vee \bar{x}$.

We write $\pi : F \vdash D$ to denote that π is a derivation of D from F ; we write $F \vdash D$ to denote that some such π exists. When D is the empty clause \perp , we call π a *resolution refutation* of F , and write $\pi : F \vdash \perp$. The *length*, or *size*, of a resolution derivation/refutation is the number of clauses in it.

A resolution derivation π can be represented as a list of clauses annotated with explanations for each clause how it was obtained. This is illustrated in Figure 7.1a for a resolution refutation of the CNF formula

$$(x \vee y) \wedge (x \vee \bar{y} \vee z) \wedge (\bar{x} \vee z) \wedge (y \vee \bar{z}) \wedge (\bar{y} \vee \bar{z}) . \tag{7.3}$$

A derivation $\pi = (D_1, D_2, \dots, D_L)$ from F can alternatively be represented by a directed acyclic graph (DAG) G_π , also referred to as a *proof DAG*, in the following way. The vertices of G_π are $\{v_1, v_2, \dots, v_L\}$, with vertex v_i labelled by the clause D_i . The sources (also called *leaves*) of the DAG are vertices labelled with the axiom clauses in F . Without loss of generality there is a unique sink and it is labelled with D_L .⁶ If π is a refutation, the sink is labelled with \perp . Every vertex that is not a source has indegree two and is the resolvent of its two predecessors. See Figure 7.1b for the refutation in Figure 7.1a represented as a DAG instead.

A resolution refutation π is *tree-like* if G_π is a tree, or equivalently, if every clause D_i in the refutation is used at most once as a hypothesis in an application

⁶Any other sinks are parts of superfluous subderivations that can be removed.

of the resolution rule. Note that it is permitted that clauses are repeated in the sequence $\pi = (D_1, D_2, \dots, D_L)$, so different vertices in G_π can be labelled by the same clause if need be (though every repetition counts towards the size of the refutation). The refutation in Figure 7.1b is not tree-like since the clause \bar{z} is used twice, but it could be made tree-like if we added a second derivation of \bar{z} from $y \vee \bar{z}$ and $\bar{y} \vee \bar{z}$. More generally, any derivation can be converted into a tree-like derivation by repeating subderivations, but possibly at the cost of an exponential increase in the size of the proof [Tse68, BSIW04].

As discussed in Section 7.2.2, *completeness* and *soundness* are fundamental properties of proof systems. For resolution, they are as follows.

Theorem 7.3.1. *Let F be a CNF formula and C be a clause.*

Completeness: *If $F \models C$, then there is a clause $C' \subseteq C$ such that $F \vdash C'$. In particular, if F is unsatisfiable, then F has a resolution refutation.*

Soundness: *If $F \vdash C$, then $F \models C$. In particular, if F has a resolution refutation, then F is unsatisfiable.*

For technical reasons it is sometimes convenient to allow also the *weakening rule*

$$\frac{B}{B \vee C}, \quad (7.4)$$

which makes it possible to infer a subsumed clause (i.e., a strictly weaker clause) from an already derived clause. Resolution with weakening is also sound and complete; in fact, completeness holds in the stronger sense that if $F \models C$, then C has a resolution-plus-weakening derivation from F .

It is not hard to show that weakening inferences in a resolution refutation can be eliminated without increasing the complexity of the proof. This also holds for all the notions of complexity that we will discuss later, including length, width, and space.

An important restricted form of resolution is *regular resolution*. A resolution refutation π is *regular* provided that no variable is resolved over more than once along any path in the DAG G_π . For example, the refutation of Figure 7.1 is not regular, since the inferences introducing the clauses \bar{z} and x lie on a common path, and both resolve on y . The soundness of regular resolution is immediate from the soundness of resolution. Regular resolution is also complete [DP60]. However, it was shown in [Urq11, AJPU07] that there is a family of CNF formulas F_n that have polynomial size resolution refutations, but are such that, letting $c = c(n)$ be the length of F_n (i.e., the number of clauses in F_n), the shortest regular resolution refutation of F_n has size $2^{c^{1-o(1)}}$. This gives an exponential separation between regular resolution and (general) resolution. In particular, regular resolution does not efficiently simulate resolution in the sense discussed in Section 7.2.2.

As already mentioned, resolution without weakening can polynomially simulate resolution. On the other hand, tree-like resolution cannot polynomially simulate resolution, or even regular resolution [Tse68].

Another interesting restriction on resolution which is relevant for CDCL SAT solvers is *trivial resolution*. A resolution derivation π is a trivial derivation of D from F if it can be written as a sequence of clauses $\pi = (D_1, \dots, D_L = D)$ such that

1. π is *input*; i.e., $D_1 \in F$, and for all $i \geq 1$ it holds that $D_{2i} \in F$ and that D_{2i+1} is the resolvent of the latest derived clause D_{2i-1} with the axiom D_{2i} .
2. π is regular, i.e., no variable is used more than once as the resolution variable.

The sequence π is a trivial resolution refutation provided D_L is \perp . Trivial resolution is sound but *not* complete. For example (and jumping ahead a bit), the formula (7.3) is not refutable by *unit propagation* (which will be described later), and thus does not have a trivial resolution refutation.

A *unit clause* is a clause of width 1. A *unit resolution* refutation is a resolution refutation in which each resolution inference has at least one unit clause as a hypothesis. Unit resolution is sound but not complete. Indeed, it is very closely connected to trivial resolution [BKS04]. For F a CNF formula and $C = a_1 \vee \dots \vee a_k$ a clause, there is a trivial resolution derivation of C from F if and only if there is a unit resolution refutation of $F \cup \{\bar{a}_1, \bar{a}_2, \dots, \bar{a}_k\}$, i.e., of F plus k many unit clauses. In particular, F has a trivial resolution refutation exactly when F has a unit resolution refutation.

7.3.1. DPLL and Conflict-Driven Clause Learning

The most successful present-day SAT solvers are based on *conflict-driven clause learning (CDCL)* [MSS99, MMZ⁺01], which, when run on unsatisfiable CNF formulas, is a search procedure for resolution proofs. CDCL algorithms have been highly refined and augmented with both practical optimizations and sophisticated inference techniques; however, they are based on four main conceptual ingredients:

1. *DPLL*, a depth-first, backtracking search procedure for satisfying assignments.⁷
2. *Unit propagation*, a method for making obvious inferences during search.
3. A *clause learning* algorithm that attempts to prune the search space by inferring (“learning”) clauses whenever the search procedure falsifies a clause (finds a “conflict”) and has to backtrack.
4. A *restart policy* for stopping a depth-first search and starting a new one.

We will discuss these four ingredients one at a time. Our treatment will be informal, without including all necessary details, as we presume that most readers are familiar with the concepts, and as our focus is on discussing connections between SAT solving and proof complexity. For a more detailed treatment, see Chapter 3 on complete algorithms and Chapter 4 on CDCL solvers.

7.3.1.1. Depth-First Search Using the DPLL Method

The first ingredient, *DPLL*, is named after the four collective authors, Davis, Putnam, Logemann and Loveland, of two of the primary sources for resolution

⁷Often unit propagation is included as part of DPLL; however, we adopt the convention that DPLL means a backtracking search, and does not need to include unit propagation. The original definition of the DPLL algorithm in [DLL62], following [DP60], included both unit propagation and the pure literal rule.

```

if the partial assignment  $\rho$  falsifies some clause of  $F$  then
  | return false;
end
if  $\rho$  satisfies  $F$  then
  | Output  $\rho$  as a satisfying assignment and terminate.
end
Pick some unassigned literal,  $a$ , called the decision literal;
Extend  $\rho$  to set  $a$  true;
Call this DPLL procedure recursively;
Update  $\rho$  to set  $a$  false;
Call this DPLL procedure recursively (again);
Update  $\rho$  to make  $a$  unassigned;
return false;

```

Figure 7.2. DPLL recursive procedure (without unit propagation).

and automated solvers, namely [DP60] and [DLL62]. The DPLL procedure is given as input a CNF formula F , i.e., a set of clauses. Its task is to determine whether F is satisfiable, and if so to find a satisfying assignment. The DPLL algorithm maintains a partial truth assignment ρ and runs the algorithm shown in Figure 7.2 recursively. Initially, ρ is the empty assignment with all variables unassigned. Each time the recursive procedure is called it chooses a *decision literal* a , and tries setting it first true and then false, using recursive calls to explore the two assignments to a . It either halts with a satisfying assignment, or eventually returns *false* from the top level if F is unsatisfiable.

There is an exact correspondence between the DPLL search procedure (without unit propagation) and tree-like, regular resolution refutations.⁸ In one direction, any tree-like regular resolution refutation π containing S occurrences of clauses corresponds to a DPLL search tree with S nodes (each node an invocation of the recursive procedure). Such a search tree can be generated by letting the DPLL search procedure traverse the proof tree G_π starting at \perp in a depth-first fashion. When the traversal of G_π is at a node labelled with a clause C , the partial assignment ρ falsifies C . Conversely, any DPLL search with S calls to the recursive procedure gives rise to a tree-like, regular resolution refutation with $\leq S$ clauses. Such a refutation can be formed from the leaves starting with clauses falsified by the deepest recursive calls. Pairs of recursive calls are joined by resolution inferences as needed (possibly with parts of the DPLL search tree being pruned away).

7.3.1.2. Unit Propagation

The second ingredient is *unit propagation*. Unit propagation can be viewed as a way to guide the DPLL search procedure, but it becomes much more important when used with clause learning, as will be discussed momentarily. Suppose that the CNF formula F contains a clause $C = a_1 \vee \dots \vee a_k$ and that the current partial assignment ρ has set all but one of the literals in C false, and that the remaining literal a_i is unassigned. Then *unit propagation* is the operation of setting a_i true

⁸Any tree-like resolution refutation can be converted to be regular by pruning away parts of the proof where variables are resolved on more than once.

```

 $\rho_0 \leftarrow \rho$ ;
Extend  $\rho$  by unit propagation for as long as possible;
if  $\rho$  falsifies some clause of  $F$  then
  |  $\rho \leftarrow \rho_0$ ;
  | return false;
end
if  $\rho$  satisfies  $F$  then
  | Output  $\rho$  as a satisfying assignment and terminate.
end
Pick some literal  $a$  not set by  $\rho$  (the decision literal);
Extend  $\rho$  to set  $a$  true;
Call this DPLL procedure recursively;
Update  $\rho$  to set  $a$  false;
Call this DPLL procedure recursively (again);
 $\rho \leftarrow \rho_0$ ;
return false;

```

Figure 7.3. The recursive procedure for DPLL with unit propagation.

and adding it to ρ . This can be done without loss of generality, since setting a_i false would falsify C .

Unit propagation is directly related to the unit resolution proof system discussed earlier. Let F' be the CNF formula consisting of F plus the unit clauses \bar{a} for all literals a such that $\rho(a) = 0$. Then repeatedly applying unit propagation starting with F and the partial assignment ρ will falsify some clause of F if and only if F' has a unit resolution refutation.

With unit propagation, the recursive procedure for the DPLL algorithm is shown in Figure 7.3. This runs similar to the algorithm of Figure 7.2, but now does unit propagation whenever possible. Values of ρ can be set either as decision literals or by unit propagation (with ρ_0 used to restore the state of ρ when returning). Each set literal is assigned a *decision level*. Literals propagated even before any decision literal has been set are assigned decision level 0. The decision level is incremented when setting a decision literal, and unit propagated literals are given the decision level of the last decision literal. Each call to the procedure either halts with a satisfying assignment extending ρ , or returns false if no such satisfying assignment exists.

7.3.1.3. Clause Learning and Nonchronological Backtracking

The third ingredient in the recipe for state-of-the-art SAT solvers is a crucial one: the use of *clause learning* to infer new clauses. This modifies the first step of DPLL algorithm to also add clauses to the formula; namely, when ρ falsifies some clause of F , i.e., reaches a *conflict*, then clause learning is used to derive one or more clauses and add them to F . The intent is that the learned clauses will be used in the future to prune the search space and avoid repeatedly exploring the same impossible assignments to literals. Essentially all clause learning methods used in practice will infer only clauses C that can be derived from F (together with previously learned clauses) by trivial resolution. This is due to the equivalence between trivial resolution refutations and unit resolution refutations and the fact

that DPLL/CDCL uses unit propagation to find contradictions.

The most popular methods for learning clauses in modern CDCL SAT solvers are based on the *first unique implication point (UIP)* learning scheme, as illustrated in Figure 7.4. But for purposes of the connection between clause learning and trivial resolution, the key point is that the learned clause is chosen to be a clause $C = a_1 \vee \dots \vee a_k$ with the following two properties:

1. C is falsified by ρ , i.e., $\rho(a_i) = 0$ for each a_i in C , and
2. there is a unit resolution refutation of F plus the k unit clauses $\{\bar{a}_i \mid i \in [k]\}$.

When C is chosen in this way, C is called a *reverse unit propagation (RUP)* clause or an *asymmetric tautology (AT)* for F .⁹ By the correspondence between trivial resolution and unit resolution, this is equivalent to the existence of a trivial resolution derivation of C from F .

A third property that typically holds for a learned clause C is the *UIP* or *assertive property*:

3. C contains exactly one literal a_i set to false at the last decision level; the rest of the literals in C were set false at earlier decision levels. The literal a_i is called the *UIP literal* or *asserting literal*.

When C is chosen so that a_i is as “close as possible to the contradiction,” then this is called 1UIP learning, and C is called the *first unique implication point (UIP) clause* with a_i being the *UIP literal*. There is always at least one UIP clause, since the decision literal at the last level is a UIP literal (because this decision led to a conflict).

In order to visualize conflict analysis, it is helpful to construct a *conflict graph* that shows how assignments to different literals causes propagations of other literals. Figure 7.4 illustrates 1UIP clause learning in the conflict graph formed from the clauses

$$\{\bar{x} \vee \bar{a} \vee z, \bar{x} \vee \bar{z} \vee y, \bar{y} \vee t, \bar{y} \vee \bar{a} \vee u, \bar{y} \vee \bar{u} \vee v, \bar{y} \vee \bar{t} \vee s, \bar{u} \vee \bar{b} \vee \bar{c} \vee w, \bar{t} \vee \bar{v} \vee \bar{w}, \bar{a} \vee \bar{b} \vee c\}. \quad (7.5)$$

Arrows indicate which literals are involved in causing which unit propagations. The three literals a, b, c have been set true at earlier decision levels, and the decision literal at the last level is x . The 1UIP learned clause is $\bar{a} \vee \bar{b} \vee \bar{c} \vee \bar{y}$, with the 1UIP literal being y . The dashed line shows the portion of the conflict graph with propagations that depend on y , and the predecessors vertices of all edges crossing this cut are the literals a, b, c , and y that form the learned clause.

For an alternative view of the same conflict analysis, Figure 7.5a shows the unit propagations caused by setting x to 1, where $x \stackrel{\text{DEC}}{\leftarrow} 1$ denotes that x is set true as a decision literal and $\ell \stackrel{C}{\leftarrow} \sigma$ denotes the literal ℓ being set to the truth value σ by unit propagation using clause C . The 1UIP clause $\bar{a} \vee \bar{b} \vee \bar{c} \vee \bar{y}$ is derived as depicted in Figure 7.5b by traversing backwards through the list of unit propagated literals starting with the clause $\bar{t} \vee \bar{v} \vee \bar{w}$ that was falsified. Each time a literal is reached that is set false in the current clause, a resolution inference

⁹See [GN03, Van08] for the original definitions of RUP; the modern terminology AT is discussed in [HJB10, JHB12].

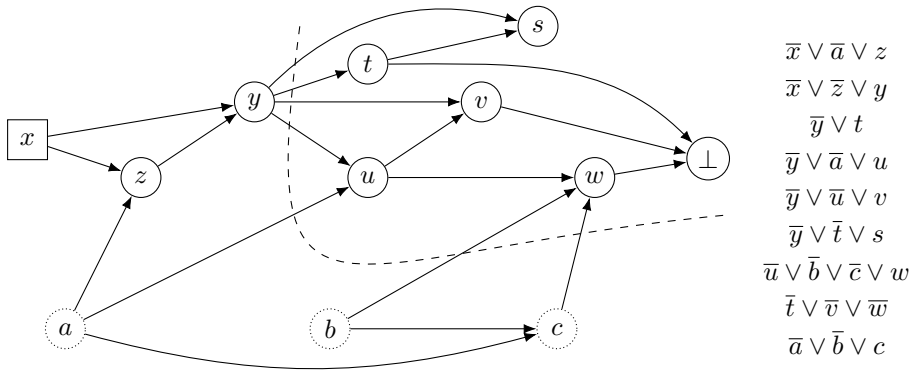


Figure 7.4. Example of a conflict graph and 1UIP learning with learned clause $\bar{a} \vee \bar{b} \vee \bar{c} \vee \bar{y}$. This conflict graph is generated using the clauses in Equation (7.5), also shown on the right in the figure.

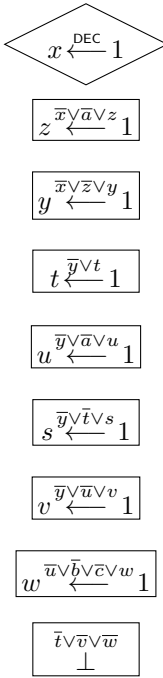
is applied to update the current clause. (Note how s is skipped as it does not appear negated in $\bar{b} \vee \bar{c} \vee \bar{t} \vee \bar{u} \vee \bar{y}$.) When we reach for the first time a clause (in this example, $\bar{a} \vee \bar{b} \vee \bar{c} \vee \bar{y}$) that contains only a single literal at the last decision level (in this example, y), we have obtained the 1UIP clause.

A nice feature of the 1UIP clause learning scheme, and the reason that the learned clause is called *asserting*, is that this clause contains exactly one literal a_i that has not been set false at an earlier decision level. This means that after the solver has backtracked and undone the last decision, the literal a_i will unit propagate to true, flipping the value it had before. For example, the clause learned in Figure 7.4 is asserting: a, b and c have been set at earlier levels, and so \bar{y} can be inferred from the learned clause by unit propagation. Conflict analysis algorithms with this property are sometimes referred to as *asserting clause learning schemes*, and learning schemes used in modern CDCL solvers tend to be of this type.

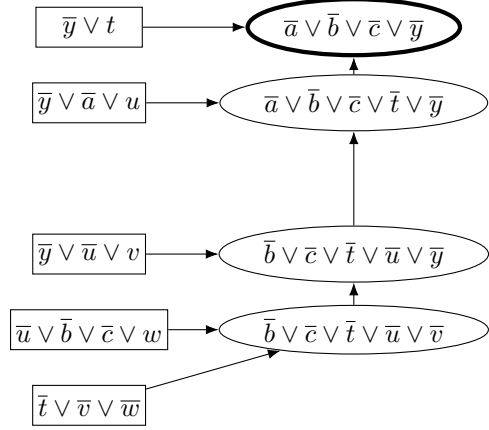
Perhaps the most direct way to incorporate both unit propagation and clause learning (but not backjumping or restarts) in DPLL is as in the algorithm shown in Figure 7.6, which we will call *DPLL with clause learning (DPLL-CL)*. We want to emphasize that DPLL-CL is not an algorithm that would be used in practice; we instead introduce it as a didactic tool to illustrate how there is a natural algorithmic development from DPLL to CDCL.

The DPLL-CL algorithm has been formulated to be faithful to the idea of DPLL as a depth-first search procedure, but has been updated with clause learning. Like the basic DPLL algorithm, it acts to set a literal a first true and then false. The first recursive call to DPLL-CL has the decision literal a set true. If that call returns, the newly learned clauses may already be enough to set a false by unit propagation. But if not, a can be set false as a decision literal anyway. For example, in the 1UIP learning in Figures 7.4 and 7.5 the learned clause allows y to be set false by unit propagation, but it does not allow \bar{x} to be derived by unit propagation. Nonetheless \bar{x} is a consequence of the literals set in ρ .¹⁰

¹⁰This will not generally be true in the CDCL algorithm described next, since CDCL allows



(a) The unit propagations triggered by setting x true.



(b) The trivial resolution derivation of $\bar{a} \vee \bar{b} \vee \bar{c} \vee \bar{y}$ (written bottom-up).

Figure 7.5. Unit propagation derivation of the contradiction shown in Figure 7.4 and trivial resolution derivation of the learned clause (from the clauses in (7.5)).

The DPLL-CL algorithm in Figure 7.6 still lacks an important component of the CDCL algorithm, namely *backjumping* (also called *nonchronological backtracking*). What this means is that the solver backtracks not just one decision level but potentially multiple decision levels at once.¹¹ For an example of a common way to use backjumping, refer to the conflict graph and 1UIP learned clause of Figures 7.4 and 7.5. There, x has been set as a decision literal with decision level $lev(x)$. The literals a , b and c were set with strictly lower decision levels $lev(a)$, $lev(b)$ and $lev(c)$; let $L \leq lev(x) - 1$ denote the maximum of these three levels. Once the 1UIP clause $\bar{a} \vee \bar{b} \vee \bar{c} \vee \bar{y}$ has been learned, the literal y can be propagated to true already at decision level L , which is called the *assertion level*. To readily take advantage of this, the CDCL algorithm uses backjumping (nonchronological backtracking) to backtrack to decision level L , which could be much smaller than $lev(x) - 1$. Another reason for backjumping is that it can be viewed a miniature version of restarting, which we will discuss next, in that a restart consists of backtracking to decision level 0.

backjumping to backtrack multiple decision levels.

¹¹It can be noted, though, that a CDCL algorithm with chronological backtracking, somewhat similar in spirit to the DPLL-CL algorithm in Figure 7.6, was recently proposed in [NR18] and has been shown to perform very well on certain types of benchmarks.

```

 $\rho_0 \leftarrow \rho;$ 
loop
  Extend  $\rho$  by unit propagation for as long as possible;
  if  $\rho$  falsifies some clause of  $F$  then
    | Optionally learn one or more clauses  $C$  and add them to  $F$ ;
    |  $\rho \leftarrow \rho_0$ ;
    | return false;
  end
  if  $\rho$  satisfies  $F$  then
    | Output  $\rho$  as a satisfying assignment and terminate.
  end
  Pick some literal  $a$  not set by  $\rho$  (the decision literal);
  Extend  $\rho$  to set  $a$  true;
  Call DPLL-CL recursively;
  Unset the value of  $a$  in  $\rho$ ;
  Extend  $\rho$  by unit propagation for as long as possible;
  if the value of  $a$  is set in  $\rho$  (if so, it is false) then
    | continue (with the next loop iteration);
  end
  Update  $\rho$  to set  $a$  false;
  Call DPLL-CL recursively (again);
   $\rho \leftarrow \rho_0$ ;
  return false;
end loop

```

Figure 7.6. The recursive procedure DPLL-CL for DPLL with clause learning.

7.3.1.4. Restarts

The fourth ingredient in our CDCL list is the use of *restarts*. A restart consists of halting the CDCL search, while preserving the clauses learned from F so far, then starting a new CDCL search. As just noted, this already fits into the CDCL algorithm described earlier, as setting the new (backjumping) decision level L' equal to 0 causes a restart. The new CDCL search can use different decision literals, and this may provide an advantage in searching for either a satisfying assignment or a refutation. In practice, the use of restarts is crucial for CDCL solver performance. As is discussed later in this section, there are also theoretical reasons why restarts may be beneficial. There are several intuitive explanations of why restarts might be useful, but ultimately, it comes down to the fact that doing a restart allows the CDCL search to try out a different choice of decision literals. There have been extensive investigations of how to do *adaptive restarts* [AS12, Bie08] and how to choose decision literals, e.g., by using *variable state independent decaying sum (VSIDS)* [MMZ⁺01], *variable move to front (VMTF)* [Rya04], and *phase saving* [PD07]. Going into details about this is far beyond the scope of this survey, but see, e.g., [BF15] for an evaluation of different decision strategies, and [AS12, BF19, Hua07] for good discussions of restart strategies.

7.3.1.5. Putting It All Together

We are now ready to present pseudo-code for an abstract formulation of the CDCL algorithm. Unlike our earlier algorithms, this one is not implemented as a recursive procedure; this is because backjumping allows backtracking multiple decision

```

L ← 0;
ρ ← empty assignment;
loop
  Extend ρ by unit propagation for as long as possible;
  if ρ satisfies F then
    | return ρ as a satisfying assignment;
  else if ρ falsifies some clause of F then
    | if L = 0 then
    |   | return “Unsatisfiable”;
    | end
    Learn one or more clauses C and add them to F;
    Let L' < L be the minimal assertion level among the clauses learned;
    Unassign all literals set at levels > L';
    L ← L';
  else
    | Pick some unassigned literal a (the decision literal);
    | L ← L + 1;
    | Extend ρ to set a true;
  end
  continue (with the next iteration of the loop);
end loop

```

Figure 7.7. The CDCL algorithm (in skeletal form and without clause deletions).

levels at once. Instead, the CDCL algorithm maintains a current decision level L . Whenever a literal b is set, either as a decision literal or by unit propagation, it is given the current decision level as its level, denoted $lev(b)$. The CDCL algorithm is shown in Figure 7.7; its input is a formula F .

Our pseudo-code does not specify how to implement clause learning. A typical implementation will learn a 1UIP clause (with some postprocessing that we do not discuss here); this clause is asserting in that it allows a new literal to be set by unit propagation. The backjump level L' is then chosen as the assertion level of this clause, i.e., the minimum decision level at which this literal can be unit propagated. The partial assignment ρ will set a literal a true at level 0 if and only if the unit clause a is in F . It follows that if the CDCL algorithm returns “Unsatisfiable,” then the current set of clauses F is unsatisfiable. Since clause learning only learns clauses that are consequences of earlier clauses, this means that also the original (input) formula F is unsatisfiable.

If the CDCL algorithm uses 1UIP clause learning, or other clause learning schemes based on the conflict graph, then the learned clauses will be derivable by trivial resolution derivations (by the correspondence between trivial resolution and unit resolution). Thus, if the CDCL algorithm returns “Unsatisfiable”, the original formula F has a resolution refutation of size polynomial in the number of clauses learned by the CDCL algorithm.

Figure 7.8b shows an example of a full CDCL refutation for the formula

$$(u \vee x \vee y) \wedge (x \vee \bar{y} \vee z) \wedge (\bar{y} \vee \bar{z}) \wedge (\bar{x} \vee z) \wedge (\bar{x} \vee \bar{z}) \wedge (\bar{u} \vee w) \wedge (\bar{u} \vee \bar{w}) \quad (7.6)$$

based on 1UIP clause learning. Decision variables are shown in diamonds. Unit propagations are shown in rectangles. Dashed diamonds and boxes indicate assignments that were preserved after the previous conflict. Darker ovals indicate

learned clauses. As shown, the CDCL search finds three conflicts. The first conflict is found after \bar{w} and \bar{x} are chosen as decision literals at decision levels 1 and 2. The resulting learned clause $u \vee x$ is asserting, so x is set true by unit propagation at level 1. The second conflict learns the unit clause \bar{x} , which is an asserting clause setting x false at level 0. The CDCL solver then performs backjumping (nonchronological backtracking); namely, it backtracks to unset all values set at decision levels above the level of the asserted literal x . In this case, backjumping involves backtracking out of decision level 1, unassigning w and u , so that w is no longer a decision literal. The third conflict arises at level 0, and completes the CDCL procedure. However, for pedagogical reasons — to get a clearer connection to the resolution proof — the illustration assumes that the solver does not terminate immediately after detecting a conflict at decision level 0, but instead performs a final round of conflict analysis to formally derive the empty clause \perp . Figure 7.8c shows the corresponding resolution refutation of F . Note that each conflict forms a trivial resolution derivation relative to the current CNF formula F as augmented with learned clauses.

7.3.1.6. Clause Deletions

It should be mentioned that one aspect of CDCL solving that is absolutely crucial in practice, but that our discussion above completely ignores, is the use of *clause deletion*, or *clause erasure*. Clause deletion means removing some of the learned clauses. This helps reduce memory usage and, even more importantly, allows unit propagation to run faster. The disadvantage is that clause erasure might delete clauses that would have been useful in the future (but results in [KN20] indicate that in some cases erasing clauses can actually help the solver find shorter resolution refutations). Some theoretical results related to clause deletion will be discussed later in the context of resolution space in Sections 7.4.3 and 7.4.4.

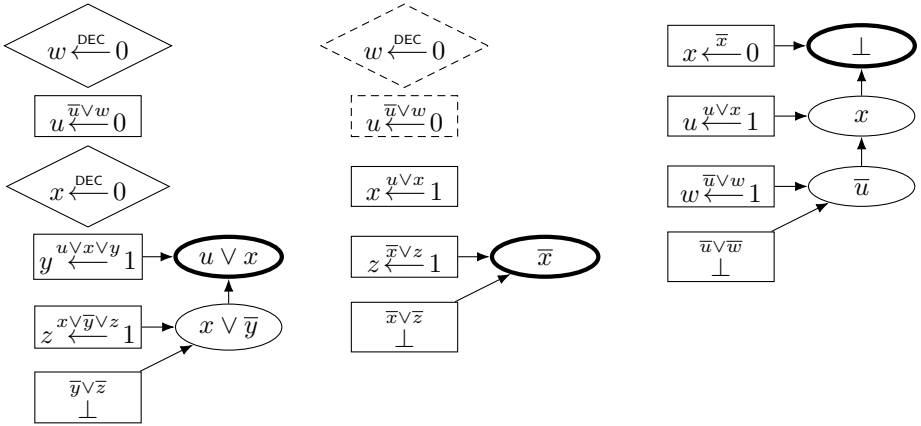
7.3.2. Proof Logging in CDCL Solvers

The output of a SAT solver given a CNF formula F will be either a satisfying assignment for F or the assertion that F is unsatisfiable. In the second case, it can be useful to not just assert that F is unsatisfiable, but also to produce a refutation of F so that this assertion can be verified. Of course, the execution of the SAT solver can serve as a kind of proof, but this is not ideal; first because there could be bugs in the design or implementation of the SAT solver, and second because it does not correspond to a useful proof system. A better option is to output a resolution refutation: this is always possible for the CDCL constructions we have discussed so far, as illustrated in Figure 7.8.

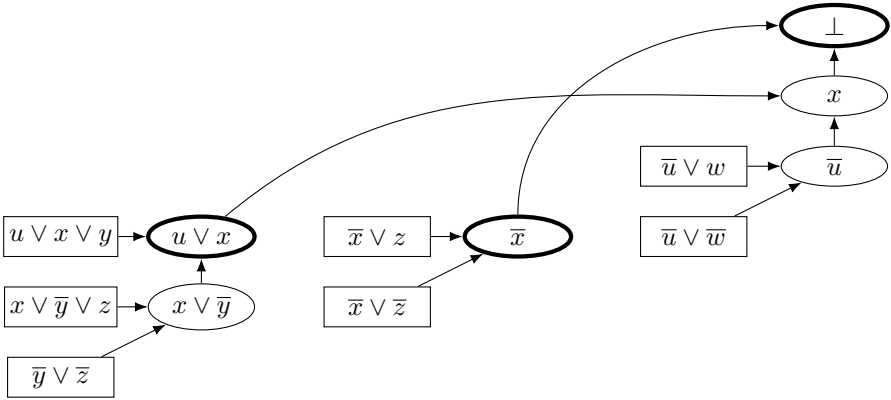
However, the fact that CDCL solvers (implicitly) produce resolution proofs does not necessarily mean that this is the best way of producing certificates of correctness. For a large formula the resolution proof can get very long, containing lots of details that are not really necessary for efficient verification. Also, forcing the solver to output all steps in every conflict analysis can incur a large overhead in running time. A way to get more concise certificates (see [GN03, Bie06, Van08, HHW13a]) is to use a *proof trace* consisting only of the sequence of learned clauses generated during the CDCL search (preferably with unneeded learned clauses

$$(\bar{u} \vee w) \wedge (u \vee x \vee y) \wedge (x \vee \bar{y} \vee z) \wedge (\bar{y} \vee \bar{z}) \wedge (\bar{x} \vee z) \wedge (\bar{x} \vee \bar{z}) \wedge (\bar{u} \vee w) \wedge (\bar{u} \vee \bar{w})$$

(a) The unsatisfiable CNF formula in (7.6).



(b) A CDCL refutation of the CNF formula in Figure 7.8a.



(c) The corresponding resolution refutation.

Figure 7.8. Part (a) shows the unsatisfiable CNF formula (7.6). Part (b) shows a complete CDCL run establishing the unsatisfiability of this CNF formula. Part (c) shows the resolution refutation corresponding to this CDCL execution.

omitted). For instance, in the example in Figure 7.8, the proof trace is just the sequence of clauses $u \vee x$, \bar{x} , \perp . Such a proof is called a *RUP proof*, since each clause follows by reverse unit propagation from F plus the preceding clauses in the sequence. The property of being RUP clause is checkable in polynomial time using unit propagation; thus the correctness of a RUP proof trace can be checked in time polynomially bounded by the size of F and the total size of the proof trace.

A very important aspect in applied SAT solving, which is also relevant for proof logging, is that extensive *preprocessing* of the input is performed before the main CDCL search algorithm starts, and some solvers, such as *Lingeling* [Lin], *CryptoMiniSat* [Cry], and *CaDiCaL* [CaD], even interleave preprocessing techniques with CDCL search; this is known as *inprocessing* [JHB12]. This involves using a number of techniques which, although known to be theoretically very bad in the worst case, are indispensable in practice. Many of these techniques can be formalized within resolution and hence in RUP proofs, but not all. Nowadays, RUP proof traces have therefore been supplanted by more sophisticated DRAT proofs [HHW13b]. DRAT proofs generalize RUP proofs, but can also simulate stronger methods of reasoning — in fact, all the way up to extended resolution,¹² as will be discussed later in Section 7.8.

7.3.3. Efficiency of CDCL Proof Search

Now that we have given formal descriptions of different methods for SAT solving, we can study the complexity of proofs as generated by these methods when they are used on unsatisfiable CNF formulas. As discussed above, the DPLL method corresponds to tree-like resolution, which can be exponentially worse than (general, DAG-like) resolution. Since CDCL is only looking for structurally very restricted proofs, it is natural to ask how efficient CDCL proof search can be compared to the best possible general resolution proof. (Henceforth, when we talk about CDCL, we mean CDCL *with restarts*, unless explicitly stated otherwise.) Note that if the formula is exponentially hard for resolution, then basic CDCL search cannot be expected to run fast since it is searching for a resolution proof. But what we can do is to benchmark the solver against the best possible outcome, i.e., the most efficient proof. Thus, we can ask whether a CDCL search strategy can implement efficient proof search in the sense that the algorithm finds a refutation which is no more than polynomially longer than, say, the shortest possible resolution refutation.

This turns out to be a deep and difficult question to answer. What is known is that there is no algorithm that can find resolution proofs efficiently unless $P = NP$ [AM20] (this recent paper strengthens previous results in [AR08, MPW19]). In formal proof complexity terminology, resolution is said to be *non-automatable*.

¹²It is important to note, however, that the power of DRAT proofs to simulate extended resolution is (at least as of yet) a purely theoretical result. Many reasoning techniques that, while powerful, fall well short of extended resolution, cannot currently be formalized in a practically efficient way in DRAT, two notable examples being cardinality reasoning and Gaussian reasoning. While it is possible in principle to handle cardinality reasoning or Gaussian reasoning in the DRAT proof system [HB18, BT19, HKB20], the attempts made so far to implement this in practice in state-of-the-art CDCL solvers have been far too inefficient.

What we can do instead is to seek bounds on the proof-theoretic strength of so-called *nondeterministic*¹³ solvers that are assumed to somehow magically make good choices for certain heuristics during the search. In a line of works including [BKS04, HBPV08, BHJ08] and culminating in the papers [PD11, AFT11], it was shown that nondeterministic CDCL can be as efficient as the resolution proof system except possibly for a polynomial blow-up.

More technically speaking, the assumptions needed are that the CDCL solver is allowed to magically choose decision variables and values to assign to these variables, and also keeps every single clause ever learned during the search. Furthermore, it has to make somewhat frequent restarts, but not more frequent than what is already standard in state-of-the-art solvers, and has to use some asserting clause learning scheme (in particular, 1UIP works fine, but a strength of the simulation is that any other asserting scheme will also be good enough). With these assumptions, the main result in [PD11] is that a CDCL solver can establish unsatisfiability of a CNF formula in time polynomial in the shortest resolution refutation of that formula. One possible way of interpreting this result is that in cases where the decision heuristic works well enough, and when the right learned clauses are kept in memory, then a CDCL solver could in principle run fast for formulas that possess short resolution proofs (although it follows from [AM20] that for any concrete choice of heuristics there are tricky formulas that will make these heuristics fail unless $P = NP$).

The construction of [PD11] showing that CDCL simulates resolution can be summarized as follows. Suppose that the formula F has a resolution refutation $\pi = (D_1, \dots, D_L = \perp)$. Intuitively, the goal for the CDCL search is to successively learn the clauses D_i for $i = 1, 2, \dots$ until it reaches $D_L = \perp$, which corresponds to a conflict at decision level 0 (as in the example in Figure 7.8). Learning exactly the clauses D_i might not always be possible, but a key insight in [PD11] is that the CDCL solver can instead learn other clauses which together are as powerful as D_i . For this, using a definition from [AFT11], we say that a set of clauses F' *absorbs* a clause $D = a_1 \vee \dots \vee a_k$ provided that setting any $k-1$ literals of D false allows either the remaining literal of D or the empty clause to be derived by unit propagation from the clauses of F' . As a small, concrete, example, the set of clauses $\{x_1 \vee y_1, \bar{y}_1 \vee x_2 \vee y_2, \bar{y}_2 \vee x_3\}$ absorbs $x_1 \vee x_2 \vee x_3$, since falsifying any two of the x_i -variables will propagate the third to true for both the clause and the clause set.

What absorption means is that it seems to an outside observer that the set of clauses F' contains D , since all unit propagations that would occur if D were in F' can be observed now as well. The construction of [PD11] allowing CDCL search to simulate the resolution refutation π is based on showing that the clause learning can work well enough to successively absorb the clauses D_1, \dots, D_L in the refutation π . Suppose that the solver has added enough learned clauses to the clause database $F' \supseteq F$ to have absorbed D_1, \dots, D_{i-1} . To absorb D_i , the CDCL solver repeatedly tries to falsify all literals in D_i . When this is no longer

¹³Note that the word “nondeterministic” is used here in the technical sense of the definition of the complexity class NP, where a computation can split into two branches at every step and succeeds in finding a solution if one of the (potentially exponentially many) branches does so. It thus has a very different meaning from “randomized,” where the requirement is that the successful branch should be found with high probability.

possible to do without the last unassigned literal in D_i being propagated, the clause has been absorbed. As long as it is possible to falsify D_i , then since the clauses resolved to obtain D_i are already absorbed by the inductive hypothesis, it follows that unit propagation will lead to a conflict. At this point, the solver can add a new learned clause to the clause database F' and restart. It can be shown that this process of falsifying D_i and adding a new learned clause can repeat at most a polynomial number of steps before D_i is absorbed.

The independent work [AFT11] obtained an alternative, more effective version of the simulation result by showing that if a formula F has a resolution refutation in bounded width (i.e., where every clause contains only a small number of literals), then CDCL using a decision strategy with enough randomness will decide F efficiently. At first sight this might not seem so impressive — after all, it is easy to see that in this case exhaustive search in bounded width also runs fast — but the point is that a CDCL solver is very far from doing exhaustive width search and does not care at all about the existence or non-existence of resolution refutations with only small clauses.

A downside of both of these results is that it is crucial for the SAT solver never to delete clauses. This is a very unrealistic assumption, since modern solvers typically throw away a majority of the clauses learned during search. It would be nice to extend the model of CDCL in [AFT11, PD11] to capture memory usage in a more realistic way, and then study the question of whether CDCL can simulate resolution efficiently with respect to both time and space. An attempt in this direction was made in [EJL⁺16], but the following problem still remains open. (We refer to Section 7.4.3 for the formal definitions of space.)

Open Problem 7.1. Suppose that F is a CNF formula that has a resolution refutation in simultaneous length L and space s . Does this imply that there is a CDCL proof of the unsatisfiability of F (as modelled in [AFT11, PD11, EJL⁺16]) in simultaneous length and space polynomial in L and s , respectively?

Another downside of [AFT11, PD11] is that the simulation of resolution by CDCL depends crucially on the choice of decision literals. However, the correct choice of decisions is made in a highly non-constructive way — namely, as discussed above, it consists of falsifying the literals in the clauses in the (unknown) shortest resolution refutation one at a time in the order they appear in the refutation, and to do this repeatedly for each clause until it is guaranteed to be absorbed. In other words, the usual heuristics for decisions (such as VSIDS and phase saving) do not come into play at all for the simulation of resolution by CDCL. Similarly, the concrete restart heuristic used plays no role — all that matters is that restarts are frequent enough. The way the construction goes, the solver makes progress towards its goal at the first conflict after every restart, but then it might not do anything useful until the next restart happens. One can show, though, that the solver also cannot do anything harmful while waiting for the next restart (but for this to hold it is crucial that the solver never deletes any learned clauses).

It is still mostly an open problem to establish theoretical results about how the commonly used CDCL heuristics contribute (or sometimes fail to contribute) to the success of CDCL search. This is challenging to do from a mathematical

point of view, however, since these heuristics are quite sophisticated and often depend on the full history of the CDCL execution so far.

For clause learning, a seemingly obvious approach would be to prioritize learning short clauses, but it seems to have been known in the SAT community that this does not work so well in practice. A theoretical explanation for this was given in [BSJ10], where it was shown that for formulas that require resolution proofs with large clauses CDCL solvers also need to learn and keep large clauses in order to run fast (i.e., it is not enough that large clauses appear as intermediate steps in the conflict analysis). Regarding clause deletion, the theoretical results in Sections 7.4.3 and 7.4.4 can be interpreted as saying that the quite aggressive deletion policies in modern CDCL solvers can incur a substantial (sometimes exponential) increase in running time, although this can only be rigorously proven for specially designed theoretical benchmark formulas. As to activity-related decision heuristics such as VSIDS [MMZ⁺01], VMTF [Rya04], or phase saving [PD07], there is very little by way of concrete theoretical results.

Open Problem 7.2. Prove simulation or separation results for resolution versus CDCL when the solver is constrained to use decision strategies such as VSIDS, VMTF, and/or phase saving.

A very recent paper [Vin20] shows, somewhat informally speaking, that there are unsatisfiable CNF formulas for which standard decision heuristics such as VSIDS and VMTF can be exponentially worse than optimal decisions. Another recent contribution [MPR20] studies the power of CDCL when the decision heuristic is replaced by a fixed ordering of the variables, but the strongest results are unfortunately for a model that is fairly far from practice when it comes to modelling unit propagation and clause learning.

An even more fundamental open problem is to understand the role of restarts for CDCL solvers and whether they add to the theoretical reasoning power. Phrased in theoretical language, the question is whether restarts are really necessary in order for CDCL to be able to simulate resolution, or whether this can be done also without restarts. To conclude this section, we briefly discuss two related approaches to studying this problem by modelling CDCL without restarts as a proof system, namely *pool resolution* [Van05] and *RegWRTI* [BHJ08].

A pool resolution refutation consists of a resolution refutation π with an associated regular depth-first traversal; in other words, there is a depth-first traversal of the DAG G_π such that at each point during the traversal the path from the current clause back to the empty clause does not resolve more than once on any given variable. The intuition is that a regular depth-first traversal corresponds to the CDCL search, so if clauses are learned as they are traversed, they do not need to be traversed again. The fact that CDCL never chooses a decision literal which is already set and that clause learning only learns clauses containing negated literals imposes a regularity condition for each path in the depth-first search. One aspect in which pool resolution seems stronger than CDCL, though, is that the depth-first traversal allows a much richer set of clauses to be “learned” than those derived by trivial resolution from the current clause database.

The RegWRTI system is similar in spirit to pool resolution, but it uses a restricted form of clause learning that (unlike pool resolution) allows learning only

clauses that can be learned using cuts in a conflict graph. Thus RegWRTI better approximates the way clause learning works in CDCL solvers. The definition of RegWRTI is a bit complicated, so we omit it here. It is open whether pool resolution or RegWRTI simulate resolution; indeed, several candidates for separation have failed to give a separation [BB12, BBJ14, BK14].

Pool resolution and RegWRTI do not fully capture CDCL without restarts, as they do not incorporate *self-subsumption* [SB09, HS09] (also called *clause minimization*) during clause learning. For an example of self-subsumption, see Figure 7.4, where the 1UIP clause $\bar{a} \vee \bar{b} \vee \bar{c} \vee \bar{y}$ can be resolved with the clause $\bar{a} \vee \bar{b} \vee c$ to yield a better learned clause $\bar{a} \vee \bar{b} \vee \bar{y}$. Since this resolves on the literal c even though c has been set false, it does not fit the framework of pool resolution or RegWRTI. It is possible that pool resolution or especially RegWRTI can simulate clause learning augmented with self-subsumption, but this is an open question.

Open Problem 7.3. Can CDCL as modelled in [AFT11, PD11, EJL⁺16], but without restarts, efficiently simulate resolution? Or is it possible to prove that resolution is stronger than such a model of CDCL, or even stronger than RegWRTI or pool resolution?

7.4. Resolution and Proof Complexity

The previous section described the close connections between the resolution proof system and SAT solvers based on conflict-driven clause learning. Thanks to these connections, one way of analysing the potential and limitations of CDCL proof search is to study resolution refutations and establish upper and lower bounds on the complexity of such refutations. A lower bound on resolution proof length gives a corresponding lower bound on CDCL execution time, since the proof search cannot run faster than the smallest resolution proof it could possibly find.¹⁴ Conversely, an upper bound on resolution refutation length points to at least the possibility of good performance by CDCL search algorithms. There are other ways to measure resolution proof complexity beside sheer length. Notably, bounding the width or space of resolution refutations can shed light on the effectiveness of different clause learning and clause erasure strategies. In this section, we review what is known about these different proof complexity measures for resolution.

7.4.1. Resolution Length

We start by recalling that the *length* (also referred to as the *size*) of a resolution refutation is the number of clauses in it, where the clauses are counted with repetitions (which is relevant for tree-like or regular refutations). In general, proof length/size is the most fundamental measure in proof complexity, and as just discussed, lower bounds for resolution length imply lower bounds on CDCL solver running time.

Any CNF formula of size N can be refuted in resolution in length $\exp(O(N))$, and there are formulas for which matching $\exp(\Omega(N))$ lower bounds are known.

¹⁴As already discussed, however, this does not take into account the effect of preprocessing techniques that cannot be formalized within the resolution proof system, but it is still the case that resolution lower bounds often imply hardness also in practice.

Let us discuss some examples of formulas known to be hard with respect to resolution length.

Our first example is the *pigeonhole principle (PHP)*, which says that “ m pigeons do not fit into n holes without sharing holes if $m > n$.” This is arguably the single most studied combinatorial principle in all of proof complexity (see [Raz02] for a survey). When written as an unsatisfiable CNF formula, this becomes the claim that, on the contrary, $m > n$ pigeons do fit into n holes. To encode this, one uses variables $p_{i,j}$ to denote “pigeon i goes into hole j ,” and write down the following clauses, where $i \neq i'$ range over $1, \dots, m$ and $j \neq j'$ range over $1, \dots, n$:

$$p_{i,1} \vee p_{i,2} \vee \dots \vee p_{i,n} \quad \text{[every pigeon } i \text{ gets a hole]} \quad (7.7a)$$

$$\bar{p}_{i,j} \vee \bar{p}_{i',j} \quad \text{[no hole } j \text{ gets two pigeons } i \neq i'] \quad (7.7b)$$

There are also variants where one in addition has “functionality” and/or “onto” axioms

$$\bar{p}_{i,j} \vee \bar{p}_{i,j'} \quad \text{[no pigeon } i \text{ gets two holes } j \neq j'] \quad (7.7c)$$

$$p_{1,j} \vee p_{2,j} \vee \dots \vee p_{m,j} \quad \text{[every hole } j \text{ gets a pigeon]} \quad (7.7d)$$

In a breakthrough result, Haken [Hak85] proved that the PHP formulas consisting of clauses (7.7a) and (7.7b) require length $\exp(\Omega(n))$ in resolution for $m = n + 1$ pigeons, and his proof can be extended to work also for the *onto FPHP formulas* consisting of all clauses (7.7a)–(7.7d).¹⁵ Later work [Raz04a, Raz03, Raz04b] has shown that all of the PHP formula variants remain hard even for arbitrarily many pigeons m , requiring resolution length $\exp(\Omega(n^\delta))$ for some $\delta > 0$ — such formulas with $m \gg n$ are referred to as *weak pigeonhole principle (WPHP) formulas*, since the claim made by the pigeonhole principle gets weaker and weaker as m increases compared to n . What all of these lower bounds mean, intuitively, is that the resolution proof system really cannot count: even faced with the preposterous claim that infinitely many pigeons can be mapped in a one-to-one fashion into a some finite number n of holes, resolution cannot refute this claim with a proof of length polynomially bounded in the number of holes.

Since pigeonhole principle formulas have size $N = \Theta(n^3)$, Haken’s lower bound is only of the form $\exp(\Omega(\sqrt[3]{N}))$ expressed in terms of formula size, however, and so does not quite match the $\exp(O(N))$ worst-case upper bound. The first truly exponential lower bound on length was obtained for *Tseitin formulas* (an example of which is shown in Figure 7.9), which encode (the negation of) the principle that “the sum of the vertex degrees in a graph is even.” Here the variables correspond to the edges in an undirected graph G of bounded degree. Every vertex in G is labelled 0 or 1 so that the sum of the vertex labels is odd.

¹⁵To generate these formulas in the standard DIMACS format used by SAT solvers, one can use the tool *CNFgen* [LENV17, CNF] with the command line `cnfgen php <m> <n>` to obtain PHP formulas with m pigeons and n holes, `cnfgen php --functional <m> <n>` for FPHP formulas with axioms (7.7c) also, `cnfgen php --onto <m> <n>` for onto PHP formulas with axioms (7.7d), and `cnfgen php --onto --functional <m> <n>` for onto FPHP formulas with all axioms clauses (7.7a)–(7.7d) discussed above. (Here and in all following *CNFgen* examples, the syntax <n> means that an actual value of n , without brackets, should be provided on the command line.)

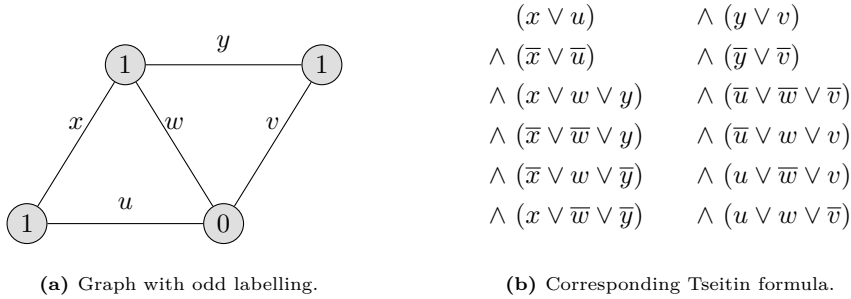


Figure 7.9. Example Tseitin formula.

The Tseitin formula for G is the CNF formula which is the conjunction of the set of clauses expressing that for each vertex of G the parity of the number of true edges incident to that vertex is equal to the vertex label. See Figure 7.9b, which displays the formula corresponding to the labelled graph in Figure 7.9a.

If we sum over all vertices, the number of all true incident edges should be odd by the construction of the labelling. However, since such a sum counts each edge exactly twice it has to be even. Thus, the Tseitin formulas are indeed unsatisfiable. Urquhart [Urq87] established that Tseitin formulas require resolution length $\exp(\Omega(N))$ if the underlying graph is a well-connected so-called *expander graph*. We cannot discuss the fascinating theory of expander graphs here, and instead refer to [HLW06] for more information, but suffice it to say that, e.g., a randomly sampled regular graph is an excellent expander *asymptotically almost surely*, i.e., with overwhelmingly large probability,¹⁶ and there are also explicit constructions. Intuitively, the lower bound in [Urq87] shows that not only is resolution unable to count efficiently in general, but it cannot even do so mod 2.¹⁷

Another example of exponentially hard formulas are **random k -CNF formulas** which are generated by randomly sampling $\Delta \cdot n$ k -clauses over n variables for some large enough constant Δ depending on k . For instance, $\Delta \gtrsim 4.5$ is sufficient to get unsatisfiable 3-CNF formulas asymptotically almost surely [DBM00] (see also Chapter 10 on random satisfiability in this handbook). Chvátal and Szemerédi [CS88] established that resolution requires length $\exp(\Omega(N))$ to refute such formulas (again asymptotically almost surely).¹⁸

By now strong lower bounds have been shown for formulas encoding **tiling**

¹⁶Formally, we say that a sequence of events \mathcal{P}_n happen *asymptotically almost surely* (sometimes also referred to as *with high probability*) if $\lim_{n \rightarrow \infty} \Pr[\mathcal{P}_n] = 1$.

¹⁷To generate unsatisfiable Tseitin formulas in DIMACS format for random 4-regular graphs over n vertices (which yields formulas that are exponentially hard for resolution asymptotically almost surely) one can use *CNFgen* with the command line `cnfgen tseitin <n>` (or, more generally, `cnfgen tseitin <n> <d>` for d -regular graphs over n vertices as long as $d \cdot n$ is even). For a standard CDCL solver without Gaussian reasoning, Tseitin formulas over random 4-regular graphs become quite hard already for around 50 vertices.

¹⁸*CNFgen* generates random k -CNF formulas with m clauses over n variables with the command line `cnfgen randkcnf <k> <n> <m>`. Random 3-CNF formulas with n variables and $\Delta \cdot n$ clauses for $\Delta = 4.5$ are noticeably hard for CDCL solvers by the time $n = 350$ or so.

problems [Ale04, DR01], **k -colourability** [BCMM05], **independent set/clique**, and **vertex cover** [BIS07], and many other combinatorial principles. For clique formulas there is an interesting range of parameters where lower bounds are *not* known, however, and so we discuss this family of formulas next.

Given a positive integer k and a graph $G = (V, E)$ with vertices V and edges E , the clique formula encodes the claim that G has a k -clique. We have variables $x_{v,i}$ with the intended meaning “vertex v is the i th member of the clique” and also think of the vertices as $V = \{1, 2, \dots, n\}$ so that we can list them in order and compare them. Letting u, v range over vertices and i, j range over clique membership indices, the formula consists of the following clauses:

$$x_{1,i} \vee x_{2,i} \vee \dots \vee x_{n,i} \quad i \in [k], \quad (7.8a)$$

$$\bar{x}_{u,i} \vee \bar{x}_{v,i} \quad u, v \in V, u < v, i \in [k] \quad (7.8b)$$

$$\bar{x}_{v,i} \vee \bar{x}_{v,j} \quad v \in V, i, j \in [k], i < j \quad (7.8c)$$

$$\bar{x}_{u,i} \vee \bar{x}_{v,j} \quad (u, v) \notin E, i, j \in [k], i \neq j \quad (7.8d)$$

$$\bar{x}_{u,j} \vee \bar{x}_{v,i} \quad u, v \in V, u < v, i, j \in [k], i < j \quad (7.8e)$$

Clauses (7.8a), (7.8b), and (7.8c) just encode that there is precisely one vertex chosen to be the i th clique member. Clauses (7.8d) enforce the constraint that there must be edges between all vertices in the clique. Finally, the clauses (7.8e) remove a potential (and artificial) source of hardness by specifying that the vertices v_1, v_2, \dots, v_k chosen as clique members $1, 2, \dots, k$ should be listed in increasing order.

From a proof complexity point of view, clique formulas become interesting when the graph G does *not* have a k -clique but when this is hard to prove. This problem can clearly be solved in time roughly n^k simply by checking if any of the $\binom{n}{k}$ many sets of vertices of size k forms a clique. Such a simple approach can be formalized already in tree-like resolution, and yields refutations of polynomial length if k is constant. A popular conjecture is that the k -clique problem must require time $n^{\Omega(k)}$ in the worst case, or even on average when graphs are sampled at random (e.g., according to the Erdős-Rényi distribution where for an n -vertex graph every single edge out of the $\binom{n}{2}$ possible ones is included uniformly and independently at random with some appropriate probability p , called the edge density).¹⁹ A natural question is whether it is possible to prove such lower bounds unconditionally for proof systems that are strong enough to capture algorithmic methods used in practice.

For tree-like resolution, an optimal $n^{\Omega(k)}$ length lower bound (i.e., optimal up to constant factors in the exponent) was established in [BGL13] for k -clique formulas on Erdős-Rényi random graphs. When k is chosen very large, on the order of n^γ for some large enough γ , $0 < \gamma < 1$, it was shown in [BIS07, Pan19] that resolution requires length scaling like $\exp(n^\delta)$ for some $\delta > 0$ for Erdős-Rényi

¹⁹To generate k -clique formulas over n -vertex random graphs for small values of k that seem likely to require resolution refutations of length $n^{\Omega(k)}$, one can use *CNFgen* with the command line `cnfgen kclique (k) gnp (n) (p)` for $p = n^{-2/(k-2)}$, which is below the threshold edge density $n^{-2/(k-1)}$ for the appearance of k -cliques and hence will yield unsatisfiable formulas asymptotically almost surely. A concrete setting of parameters that seem to yield formulas with hardness that scales nontrivially with the number of vertices n in practice for CDCL solvers is $k = 10$ and $p = n^{-1/4}$.

random graphs. For smaller k , an optimal $n^{\Omega(k)}$ length lower bound was obtained for the restricted subsystem of *regular resolution* in [ABdR⁺18], but for general resolution it remains an open problem even to establish worst-case lower bounds for small k .

Open Problem 7.4. Prove average-case resolution length lower bounds $n^{\Omega(k)}$ for k -clique formulas over Erdős-Rényi random graphs with appropriate parameters, or worst-case length lower bounds for any family of graphs.

This problem is also of intrinsic interest in proof complexity, since these formulas do not seem to be amenable to the standard techniques for proving resolution lower bounds such as the *interpolation method* [Kra97, Pud97], *random restrictions* [BP96b], or the *length-width lower bound* in [BSW01]. We will discuss some of these techniques later in the survey.

We conclude our discussion of resolution length by mentioning one slightly more recent addition to the long list of challenging combinatorial formulas already mentioned, namely the *subset cardinality formulas* studied in [Spe10, VS10, MN14] (also known as *zero-one design* or *s-gen* formulas).

To construct these formulas, we start with an $n \times n$ $(0, 1)$ -matrix with 4 non-zero entries in each row and column except that one extra non-zero entry is added to some empty cell (as in Figure 7.10a, where the extra 1 in the bottom row is in bold face). The variables of the formula are the non-zero entries of the matrix, yielding a total of $4n + 1$ variables. For each row of 4 ones in the matrix, we write down the natural 3-CNF formula encoding the *positive cardinality constraint* that at least 2 variables must be true (as in the first set of clauses in Figure 7.10b), and for the row with 5 ones the 3-CNF formula encoding that a strict majority of 3 variables must be true. For the columns we instead encode *negative cardinality constraints* that a majority of the variables in each column must be false (see the last set of clauses in Figure 7.10b). The formula consisting of the conjunction of all these clauses must be unsatisfiable, since a strict majority of the variables cannot be true and false simultaneously. We will have reason to return to these formulas below when we discuss connections between CDCL and resolution, and also when discussing cutting planes and pseudo-Boolean solving.

It was shown empirically in [Spe10, VS10] that these formulas are very challenging for CDCL solvers, but there was no analysis of the theoretical hardness. Such an analysis was provided by [MN14], where it was established that subset cardinality formulas are indeed exponentially hard if the underlying matrix is an expander (informally, if every small-to-medium set of rows has non-zero entries in many distinct columns).²⁰

7.4.2. Resolution Width

A second complexity measure in resolution, which is almost as well studied as length, is the *width* of refutations measured as the size of a largest clause in a

²⁰ *CNFgen* generates subset cardinality formulas for random $n \times n$ matrices — which are expanding, and so yield formulas that are exponentially hard for resolution, asymptotically almost surely — with the command line `cnfgen subsetcard <n>`. These formulas get noticeably hard for CDCL solvers around $n = 25$.

$$\begin{array}{l}
 \left(\begin{array}{cccccccc}
 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\
 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\
 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\
 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\
 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1
 \end{array} \right)
 \end{array}$$

(a) Matrix with row and column constraints.

$$\begin{array}{l}
 (x_{1,1} \vee x_{1,2} \vee x_{1,4}) \\
 \wedge (x_{1,1} \vee x_{1,2} \vee x_{1,8}) \\
 \wedge (x_{1,1} \vee x_{1,4} \vee x_{1,8}) \\
 \wedge (x_{1,2} \vee x_{1,4} \vee x_{1,8}) \\
 \vdots \\
 \wedge (\bar{x}_{4,11} \vee \bar{x}_{8,11} \vee \bar{x}_{10,11}) \\
 \wedge (\bar{x}_{4,11} \vee \bar{x}_{8,11} \vee \bar{x}_{11,11}) \\
 \wedge (\bar{x}_{4,11} \vee \bar{x}_{10,11} \vee \bar{x}_{11,11}) \\
 \wedge (\bar{x}_{8,11} \vee \bar{x}_{10,11} \vee \bar{x}_{11,11})
 \end{array}$$

(b) Cardinality constraints in CNF.

Figure 7.10. Matrix and (fragment of) corresponding subset cardinality formula.

resolution refutation. It is clear that the width needed to refute a formula is never larger than the number of variables n , which is in turn less than the total formula size N . It is also easy to see that an upper bound w on resolution width implies an upper bound $O((3n)^w)$ on resolution length, simply because the total number of distinct clauses of width at most w over n variables is less than $(3n)^w$. Incidentally, this simple counting argument turns out to be essentially tight, in that there are k -CNF formulas refutable in width w that require resolution length $n^{\Omega(w)}$, as shown in [ALN16].

Much less obviously, however, and much more interestingly, strong enough width lower bounds imply strong length lower bounds. Ben-Sasson and Wigderson [BSW01] (using methods based on [CEI96, IPS99]) showed that for a k -CNF formula over n variables it holds that

$$\text{refutation length} \geq \exp \left(\Omega \left(\frac{(\text{refutation width} - k)^2}{n} \right) \right) \quad (7.9)$$

where “refutation length” and “refutation width” mean the minimum length and minimum width, respectively, of any resolution refutations of the formula. (Note that these two minima could potentially be realized by different refutations.) The inequality (7.9) implies that if one can prove that a formula requires width $\omega(\sqrt{n \log n})$, this immediately yields a superpolynomial length lower bound, and a width lower bound $\Omega(N)$ in terms of the formula size N (which is lower-bounded by the number of variables n) implies a truly exponential $\exp(\Omega(N))$ length lower bound. Almost all known lower bounds on resolution length can be derived via width lower bounds in this way (in particular, essentially all the bounds discussed in Section 7.4.1²¹ although the ones predating [BSW01] were originally not obtained in this way).

For tree-like resolution refutations of k -CNF formulas, the paper [BSW01]

²¹To be precise, the exceptions are [Ale04, DR01, Raz04a, Raz03, Raz04b], where the number of variables n , and hence the formula size N , is at least as large as the refutation width squared, and where other methods must therefore be used to prove lower bounds on resolution length.

proved a sharper version

$$\text{tree-like refutation length} \geq 2^{\text{refutation width} - k} \quad (7.10)$$

of the bound in (7.9) for general resolution. This means that for tree-like resolution, even width lower bounds $\omega(\log N)$ yield superpolynomial length lower bounds. For general resolution, however, a width lower bound even as large as $\Omega(\sqrt{n \log n})$ does not imply any length lower bound according to (7.9). This raises the question of whether it is possible to improve the analysis so that (7.9) can be strengthened to something closer to (7.10) also for general resolution. Bonet and Galesi [BG01] showed that this is not the case by studying another interesting combinatorial benchmark formula, which we describe next.

The **ordering principle** says that “every finite (partially or totally) ordered set $\{e_1, \dots, e_n\}$ has a minimal element.” To encode the negation of this statement in CNF, we use variables $x_{i,j}$ to denote “ $e_i < e_j$ ” and write down the following clauses (for $i \neq j \neq k \neq i$ ranging over $1, \dots, n$):

$$\bar{x}_{i,j} \vee \bar{x}_{j,i} \quad [\text{anti-symmetry; not both } e_i < e_j \text{ and } e_j < e_i] \quad (7.11a)$$

$$\bar{x}_{i,j} \vee \bar{x}_{j,k} \vee x_{i,k} \quad [\text{transitivity; } e_i < e_j \text{ and } e_j < e_k \text{ implies } e_i < e_k] \quad (7.11b)$$

$$\bigvee_{1 \leq i \leq n, i \neq j} x_{i,j} \quad [e_j \text{ is not a minimal element}] \quad (7.11c)$$

One can also add axioms

$$x_{i,j} \vee x_{j,i} \quad [\text{totality; either } e_i < e_j \text{ or } e_j < e_i] \quad (7.11d)$$

to specify that the ordering has to be total.²² This yields a formula over $\Theta(n^2)$ variables of total size $N = \Theta(n^3)$. We remark that variants of ordering principle formulas also appear under the names **least number principle formula** and **graph tautology formula** in the literature.

It was conjectured in [Kri85] that these formulas should be exponentially hard for resolution, but Stålmarck [Stå96] showed that they are refutable in length $O(N)$ (even without the clauses (7.11d)).

As the formula is described above, it does not really make sense to ask about the refutation width, since already the axiom clauses (7.11c) have unbounded width. However, one can convert the formula to 3-CNF by applying the transformation from (7.1a) to (7.1b) to the wide axioms (7.11c), and for this version of the formula [BG01] established a width lower bound $\Omega(\sqrt[3]{N})$ (which is tight, and holds even if the axioms (7.11d) are also added). This shows that even polynomially large resolution width does not necessarily imply any length lower bounds for general resolution (and, in view of (7.10), it provides an exponential separation in proof power between general and tree-like resolution, although even stronger separations are known [BSIW04]).

²² *CNFgen* generates the ordering principle formula for a set of n elements with the command `cnfgen op <n>` and also adds totality axioms with `cnfgen op --total <n>`. These formulas are easy in practice for CDCL solvers that use VSIDS with a small enough decay factor or VMTF.

7.4.3. Resolution Space

The study of the space complexity of proofs, which was initiated in the late 1990s, was originally motivated by considerations of SAT solver memory usage, but has also turned out to be of intrinsic interest for proof complexity. Space can be measured in different ways — here we focus on the most well studied measure of *clause space*, which is the maximal number of clauses needed in memory while verifying the correctness of a resolution refutation.²³ Thus, in what follows below “space” will always mean “clause space.”

The space usage of a resolution refutation at step t is the number of clauses at steps $\leq t$ that are used at steps $\geq t$. Returning to our example resolution refutation in Figure 7.1, the space usage at step 8 is 5 (the clauses in memory at this point are clauses 1, 2, 6, 7, and 8). The space of a proof is obtained by measuring the space usage at each step in the proof and taking the maximum. Phrased differently, one can view the formula as being stored in read-only input memory, from where the axiom clauses can be read into working memory. The resolution rule can only be applied to clauses currently in working memory, and if a clause has been erased from working memory, then it is gone, and will have to be rederived if it is to be used again (or read again from the read-only input, in the case of axiom clauses). Then space measures how many clauses are used in working memory to perform the resolution refutation. Incidentally, it is not hard to see that the proof in Figure 7.1 is not optimal when it comes to minimizing space. We could do the same refutation in space 4 instead by processing the clauses in the order 4, 5, 6, 3, 7, 2, 8, 1, 9, 10. (In fact, it is even conceivable that if minimizing space is all we care about, then it might be beneficial to forget clauses and rederive them later, even if it means repeating the same steps in the resolution refutation multiple times. This indeed turns out to be the case, as discussed in Section 7.4.4 below.)

Perhaps somewhat surprisingly, any unsatisfiable CNF formula of size N can always be refuted in resolution space at most $N + O(1)$ as shown by [ET01],²⁴ though the resolution refutation thus obtained might have exponential length. Lower bounds on space were shown for pigeonhole principle formulas and Tseitin formulas in [ABSRW02, ET01] and for random k -CNF formulas in [BSG03]. For the latter two formula families the (optimal linear) lower bounds matched exactly previously known width lower bounds, and also the proof techniques had a very similar flavour. This led to the question of whether there was some deeper connection waiting to be discovered. In a very elegant paper, Atserias and Dal-

²³Note, though, that this measure underestimates the actual memory usage, since storing a clause requires more than a constant amount of memory (this is similar to how we ignore the size of clauses when defining the size of resolution proofs to be the total number of clauses in the proof). For completeness, we mention that there is also a measure *total space*, counting the total number of literals in memory (with repetitions), which has been studied in, e.g., [ABSRW02, BGT14, BBG⁺17, Bon16, GKT19].

²⁴This space upper bound can also be obtained simply by running CDCL (or even DPLL) as described in Section 7.3.1 with some arbitrary but fixed variable ordering, with the (non-standard) modification that after each backjump all clauses that are no longer needed to explain the propagation of literals in the current assignment are immediately erased.

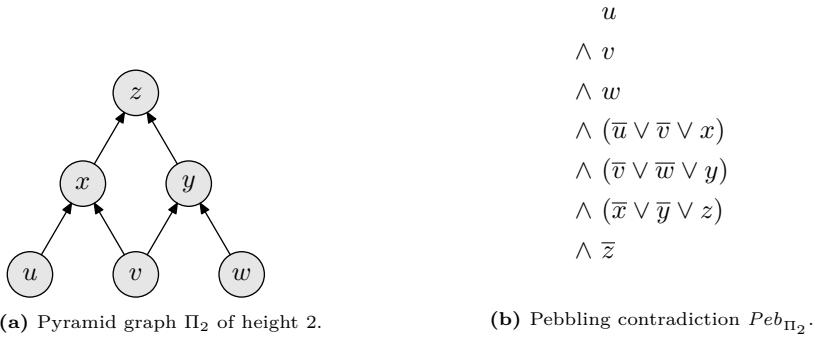


Figure 7.11. Example pebbling contradiction for the pyramid of height 2.

mau [AD08] confirmed this by showing that the inequality

$$\text{refutation space} \geq \text{refutation width} - k \quad (7.12)$$

holds for resolution refutations of k -CNF formulas. The proof of (7.12) is beautiful but uses a somewhat non-explicit argument based on finite model theory. A more explicit proof, which works by simple syntactic manipulations to construct a small-width refutation from a small-space refutation, was presented in [FLM⁺15].

Since for all formulas studied up to [AD08] the width and space complexity measures turned out to actually coincide, it is natural to ask whether (7.12) can be strengthened to an asymptotic equality. The answer to this question is negative. As shown in the sequence of works [Nor09a, NH13, BSN08], there are formulas that can be refuted in width $O(1)$ and length $O(N)$ but require space $\Omega(N/\log N)$ (i.e., formulas that are maximally easy for width but exhibit worst-case behaviour for space except for a log factor, and this result is tight since it can be shown (using a result from [HPV77] that is, incidentally, very related to the next topic of *pebbling*) that any formula refutable in length $O(N)$ can also be refuted in space $O(N/\log N)$).

The formulas used to obtain the separation result just described are **pebbling contradictions** (also called **pebbling formulas**) encoding so-called *pebble games* on bounded fan-in DAGs, which for the purposes of this discussion we additionally require to have a unique sink. In the “vanilla version” of the formula (illustrated in Figure 7.11), there is one variable associated to each vertex and clauses encoding that

- the source vertices are all true;
- if all immediate predecessors are true, then the successor vertex is true;
- but the sink is false.

There is an extensive literature on pebbling space and time-space trade-offs from the 1970s and 80s, with Pippenger [Pip80] and Savage [Sav98, Chapter 10] giving excellent overviews of some classic results in the area. Some more recent developments are covered in the upcoming survey [Nor20]. Pebbling contradic-

tions have been useful before in proof complexity in various contexts, e.g., in [RM99, BEGJ00, BSW01]. Since pebbling contradictions can be shown to be refutable in constant width but there are graphs for which the pebble game requires large space, one could hope that the pebbling properties of such DAGs would somehow carry over to resolution refutations of pebbling formulas and help us separate space and width.

Unfortunately, this hope cannot possibly materialize — a quick visual inspection of Figure 7.11b reveals that this is a Horn formula (i.e., having at most one positive literal in each clause), and such formulas are maximally easy for length, width, and space since they are decided by unit propagation. However, we can modify these formulas by substituting for every variable x an exclusive or $x_1 \oplus x_2$ of two new variables, and then expand to CNF in the canonical way to get a new formula. This process is called *XOR-ification* or *XOR-substitution* and is perhaps easiest to explain by example. Performing this substitution in the clause

$$\bar{x} \vee y \tag{7.13a}$$

we obtain the formula

$$\neg(x_1 \oplus x_2) \vee (y_1 \oplus y_2) , \tag{7.13b}$$

which when expanded out to CNF becomes

$$\begin{aligned} & (x_1 \vee \bar{x}_2 \vee y_1 \vee y_2) \\ & \wedge (x_1 \vee \bar{x}_2 \vee \bar{y}_1 \vee \bar{y}_2) \\ & \wedge (\bar{x}_1 \vee x_2 \vee y_1 \vee y_2) \\ & \wedge (\bar{x}_1 \vee x_2 \vee \bar{y}_1 \vee \bar{y}_2) . \end{aligned} \tag{7.13c}$$

As another example, applying XOR-substitution to Figure 7.11b yields the formula in Figure 7.12.²⁵

Using such XOR-substitution, it turns out that the pebbling contradiction inherits the time-space trade-offs of the pebbling DAG in terms of which it is defined [BSN08, BSN11] (and there is nothing magical with XOR — this can be shown to work also for substitution with other Boolean functions that have the right properties). Now the strong space-width separation described above is obtained by plugging in the pebbling DAGs studied in [PTC77, GT78].

7.4.4. Resolution Trade-offs

In the preceding sections, we have seen that for all the complexity measures of length, width, and space there are formulas which are maximally hard for these measures. Suppose, however, that we are given a formula that is guaranteed to be *easy* for two or more of these measures. Can we then find a resolution refutation that optimizes these complexity measures simultaneously? Or are there trade-offs, so that minimizing one measure must cause a sharp increase in the other measure? Such questions about trade-offs have a long history in computational

²⁵To generate pebbling formulas as in Figure 7.11b for pyramid graphs of height h , run *CNFgen* with the command line `cnfgen peb pyramid (h)`. Doing `cnfgen peb pyramid (h) -T xor 2` yields XOR-ified pebbling formulas as illustrated in Figure 7.12.

$$\begin{array}{ll}
(u_1 \vee u_2) & \wedge (v_1 \vee \bar{v}_2 \vee \bar{w}_1 \vee w_2 \vee y_1 \vee y_2) \\
\wedge (\bar{u}_1 \vee \bar{u}_2) & \wedge (v_1 \vee \bar{v}_2 \vee \bar{w}_1 \vee w_2 \vee \bar{y}_1 \vee \bar{y}_2) \\
\wedge (v_1 \vee v_2) & \wedge (\bar{v}_1 \vee v_2 \vee w_1 \vee \bar{w}_2 \vee y_1 \vee y_2) \\
\wedge (\bar{v}_1 \vee \bar{v}_2) & \wedge (\bar{v}_1 \vee v_2 \vee w_1 \vee \bar{w}_2 \vee \bar{y}_1 \vee \bar{y}_2) \\
\wedge (w_1 \vee w_2) & \wedge (\bar{v}_1 \vee v_2 \vee \bar{w}_1 \vee w_2 \vee y_1 \vee y_2) \\
\wedge (\bar{w}_1 \vee \bar{w}_2) & \wedge (\bar{v}_1 \vee v_2 \vee \bar{w}_1 \vee w_2 \vee \bar{y}_1 \vee \bar{y}_2) \\
\wedge (u_1 \vee \bar{u}_2 \vee v_1 \vee \bar{v}_2 \vee x_1 \vee x_2) & \wedge (x_1 \vee \bar{x}_2 \vee y_1 \vee \bar{y}_2 \vee z_1 \vee z_2) \\
\wedge (u_1 \vee \bar{u}_2 \vee v_1 \vee \bar{v}_2 \vee \bar{x}_1 \vee \bar{x}_2) & \wedge (x_1 \vee \bar{x}_2 \vee y_1 \vee \bar{y}_2 \vee \bar{z}_1 \vee \bar{z}_2) \\
\wedge (u_1 \vee \bar{u}_2 \vee \bar{v}_1 \vee v_2 \vee x_1 \vee x_2) & \wedge (x_1 \vee \bar{x}_2 \vee \bar{y}_1 \vee y_2 \vee z_1 \vee z_2) \\
\wedge (u_1 \vee \bar{u}_2 \vee \bar{v}_1 \vee v_2 \vee \bar{x}_1 \vee \bar{x}_2) & \wedge (x_1 \vee \bar{x}_2 \vee \bar{y}_1 \vee y_2 \vee \bar{z}_1 \vee \bar{z}_2) \\
\wedge (\bar{u}_1 \vee u_2 \vee v_1 \vee \bar{v}_2 \vee x_1 \vee x_2) & \wedge (\bar{x}_1 \vee x_2 \vee y_1 \vee \bar{y}_2 \vee z_1 \vee z_2) \\
\wedge (\bar{u}_1 \vee u_2 \vee v_1 \vee \bar{v}_2 \vee \bar{x}_1 \vee \bar{x}_2) & \wedge (\bar{x}_1 \vee x_2 \vee y_1 \vee \bar{y}_2 \vee \bar{z}_1 \vee \bar{z}_2) \\
\wedge (\bar{u}_1 \vee u_2 \vee \bar{v}_1 \vee v_2 \vee x_1 \vee x_2) & \wedge (\bar{x}_1 \vee x_2 \vee \bar{y}_1 \vee y_2 \vee z_1 \vee z_2) \\
\wedge (\bar{u}_1 \vee u_2 \vee \bar{v}_1 \vee v_2 \vee \bar{x}_1 \vee \bar{x}_2) & \wedge (\bar{x}_1 \vee x_2 \vee \bar{y}_1 \vee y_2 \vee \bar{z}_1 \vee \bar{z}_2) \\
\wedge (v_1 \vee \bar{v}_2 \vee w_1 \vee \bar{w}_2 \vee y_1 \vee y_2) & \wedge (z_1 \vee \bar{z}_2) \\
\wedge (v_1 \vee \bar{v}_2 \vee w_1 \vee \bar{w}_2 \vee \bar{y}_1 \vee \bar{y}_2) & \wedge (\bar{z}_1 \vee z_2)
\end{array}$$

Figure 7.12. Pebbling contradiction in Figure 7.11b with XOR-substitution.

complexity theory, but it seems that Ben-Sasson [BS09] was first to raise the issue in the context of proof complexity.

It should be noted that this kind of trade-off questions need to be phrased slightly more carefully in order to be really interesting. As observed in [Nor09b], it is often possible to prove trade-off results simply by gluing together two formulas F and G over disjoint sets of variables which have different proof complexity properties, and then obtain a trade-off result from the fact that any proof of unsatisfiability has to refute either F or G . In order to eliminate such examples and obtain formulas that have inherent trade-off properties, we can additionally require that the formulas in question should be minimally unsatisfiable, i.e., that if any clause in the formula is removed, then the residual formula is satisfiable. For most of the trade-off results we consider here, the formulas are of this flavour. Also, it can be noted that for the strongest trade-off results discussed below, the trick of gluing together disjoint formulas cannot yield trade-offs with so strong parameters anyway.

The first trade-off result in proof complexity seems to have been obtained in [BS09], where a strong space-width trade-off was established. Namely, there are formulas for which

- there are refutations in width $O(1)$;
- there are also refutations in space $O(1)$;
- but optimizing one measure causes (essentially) worst-case behaviour for the other measure, in that the product of the width and the space for any refutation must be $\Omega(N/\log N)$ (where N is the size of the formula, and the number of variables is also $\Theta(N)$).

This holds for the “vanilla version” of the pebbling contradictions in Figure 7.11b

(if one again uses the graphs studied in [PTC77, GT78]). Using techniques from [Raz16a], this space-width trade-off was strengthened in [BN20] to show that there are formulas over n variables for which resolution refutations in width w require space almost n^w , i.e., far above the linear worst-case upper bound for space.

Regarding trade-offs between length and space, it was shown in [BSN11, BBI16, BNT13] that there are formulas that can be refuted in short length and also in small space, but where even slightly optimizing one of these complexity measures causes a dramatic blow-up for the other measure. One way of obtaining such results is to take DAGs with strong pebbling time-space trade-offs (as in, e.g., [CS80, CS82, LT82, Nor12]) and consider substituted pebbling formulas (as in Figure 7.12) over such DAGs. To give an example of such a result from [BSN11], there are CNF formulas F_n of size $\Theta(n)$ and constants $\kappa' \ll \kappa$ such that:

- The formulas F_n have resolution refutations in space $\kappa' \cdot n / \log n$.
- It is also possible to refute F_n in resolution in length $O(n)$ and space $O(n)$ simultaneously.
- However, any resolution refutation of F_n in space at most $\kappa \cdot n / \log n$ has length $\exp(n^{\Omega(1)})$.

Another example of formulas exhibiting length-space trade-offs are Tseitin formulas over long, narrow rectangular grids. Building on [BBI16], it was shown in [BNT13] that there are formula families $\{F_{n,w}\}$, with $1 \leq w \leq n^{1/4}$, which are of size $\Theta(n)$ and have the following properties:

- The formulas F_n have resolution refutations in space $O(w \log n)$.
- It is also possible to refute F_n in length $n^{O(1)}2^w$ and space $2^w + n^{O(1)}$.
- But for any resolution refutation of F_n in space s the refutation length is lower-bounded by $\left(\frac{2^{\Omega(w)}}{s}\right)^{\Omega\left(\frac{\log \log n}{\log \log \log n}\right)}$.

One interesting aspect of this second result is that if we choose $w = \kappa \log n$ for some suitably large constant κ , then it follows that short refutations of these formulas require even superlinear space.²⁶ That is, although we know that any CNF formula can be refuted in linear space, short refutations can require space that is much larger than this worst-case upper bound. (It can also be noted that both of the above results are actually slightly stronger than as described here, but we omit the full technical details in order to give simpler statements.)

What a length-space trade-off result like the two examples above says is that if a resolution is of short length, then at some point during the refutation a lot of space is being used. Such results do not a priori rule out, though, that this high space usage could be an isolated spike, and that most of the time the refutation uses very little space. A more recent work [AdRNV17] established more robust trade-offs between length and *cumulative space*, exhibiting formulas where any

²⁶To generate such Tseitin formulas, run *CNFgen* with the command line `cnfgen tseitin randomodd grid (N) (M)` for N scaling like $\kappa \log(M)$ for some suitably large constant κ . It appears, though, that these formulas are just too hard to be solved by CDCL SAT solvers regardless of memory management, and so even though these formulas could potentially provide a time-space trade-off for SAT solvers this is more of a theoretical notion than something that could be observed in practice.

short proof has to use a lot of space throughout the resolution refutation.

For length versus width, we know that short refutation length implies small refutation width by (7.9). The proof of this inequality works by transforming a given short refutation into a narrow one, but the length blows up exponentially in the process. Thapen [Tha16] showed that this blow-up is unavoidable by exhibiting formulas for which there exist resolution refutations in short length, but for which any refutation in width as guaranteed by (7.9) has to be exponentially long. These formulas are slightly tricky to describe, however, and so we do not do so here. A technical issue with Thapen’s result is that for all other trade-offs discussed above there are k -CNF formulas (for $k = O(1)$) that exhibit this behaviour, but Thapen’s formulas have clauses of logarithmic width. A resolution length-width trade-off result as in [Tha16] but for k -CNF formulas was recently obtained in [LNSS20]. We also want to mention in this context that in a very intriguing work Razborov [Raz16a] obtained doubly exponential length-width trade-offs in tree-like resolution (this is measured in the number of variables in the formulas, which have exponential size and polynomial width).

7.4.5. Theoretical Complexity Measures and Hardness in Practice

The next topic we wish to discuss is whether practical hardness for CDCL is in any way related to the complexity measures of resolution length, width, and space. One interesting observation in this context is that it follows from the results reviewed in Section 7.4 — if we “normalize” length by taking a logarithm, since it can be exponential in the formula size N whereas the worst-case upper bounds for width and space are linear — that for any k -CNF formula the inequalities

$$\log(\text{refutation length}) \lesssim \text{refutation width} \lesssim \text{refutation space} \quad (7.14)$$

hold. Thus, length, width, and space form an hierarchy of increasingly strict hardness measures. Let us briefly discuss the measures again in this light:

- We know that length provides a lower bound on CDCL running time²⁷ and that CDCL polynomially simulates resolution [PD11]. However, the results in [AM20, AR08, MPW19] suggest that short resolution proofs should be intractable to find in the worst case.
- Regarding width, searching for proofs in small width is apparently a well-known heuristic in the AI community, and [AFT11] proved that CDCL has the potential to run fast if such proofs exist. (Also, note that for formulas with resolution proofs in constant width the impossibility result in [AM20] does not apply, since for such formulas exhaustive search for small-width resolution proofs runs in polynomial time.)
- As to space, memory consumption is an important bottleneck for SAT solvers in practice, and space complexity results provide lower bounds on CDCL clause database size. One downside of this is that the bounds can be at most linear, and the solver would certainly use a linear amount of memory just to store the input. However, it is important to note that the

²⁷Again, except if some non-resolution-based preprocessing or inprocessing techniques happen to be very successful.

space lower bounds can be shown to hold even in a model where we only charge for clauses in addition to the input clauses, and since it is a proof complexity lower bound it applies even to solvers which would somehow magically know *exactly* which clauses they need to keep. It could therefore be argued that in reality probably much more memory than this bare minimum should be needed.

Are width or even space lower bounds relevant indicators of CDCL hardness? Or could it be true in practice that CDCL does essentially as well as resolution with respect to length/running time? These are not mathematically well-defined questions, since state-of-the-art CDCL solvers are moving targets, but perhaps it could still be possible to perform experiments and draw interesting conclusions? Such an approach was proposed in [ABLM08], and the paper [JMNv12] reports on what seems to have been the first systematic attempt to implement this program.

In view of the discussion above it seems too optimistic that length complexity should be a reliable indicator of CDCL hardness. The work in [JMNv12] was therefore focused on comparing width and space by running extensive experiments on formulas with constant width complexity (and linear length complexity) but varying space complexity to see whether running time correlated with space. These experiments produced lots of interesting data, but it seems fair to say that the results are inconclusive. For some families of formulas the correlation between running time and space complexity looks very nice, but for other formulas the results seem quite chaotic.

One comment worth making here is that the experiments in [JMNv12] only considered the worst-case space complexity of formulas. It might be that it would be more relevant to study formulas exhibiting length-space trade-offs as in [BSN11, BBI16, BNT13] or cumulative space lower bounds as in [AdRNV17]. Another issue that should be pointed out is that formulas with low width complexity and varying space complexity are hard to find — pretty much the only known examples are the substituted pebbling formulas discussed in Section 7.4.3. Thus, it is not even clear whether the experiments measured differences in width and space complexity or some other property specific to these particular formulas. This problem seems inherent, however. One cannot just pick arbitrary benchmark formulas and compute the width and space complexity for them before running experiments, since deciding width is EXPTIME-complete [Ber12] and deciding space appears likely to be PSPACE-complete.

7.4.6. Using Theory Benchmarks to Shed Light on CDCL Heuristics

Although modern CDCL solvers are routinely used to solve real-world instances with hundreds of thousands or even millions of variables, it seems fair to say that it is still very poorly understood how these solvers can be so unreasonably effective. As should be clear from the description in Section 7.3, the basic architecture of CDCL solvers is fairly simple, but the secret behind the impressive performance of state-of-the-art solvers lies in a careful implementation of the basic CDCL algorithm with highly optimized data structures, as well in the use of dozens of sophisticated heuristics.

Unfortunately, many of these heuristics interact in subtle ways, which makes

it hard to assess their relative importance. A natural approach to gain a better understanding would be to collect “real-world benchmarks” and run experiments on an instrumented CDCL solver with different parameter settings to investigate how they contribute to overall performance, as proposed in [LMS02, KSM11], or even to study the resolution proofs corresponding to the solver executions [Sim14]. It seems quite tricky to implement this idea in a satisfactory way, however. The problem is that set of available benchmarks is somewhat limited, and is also a highly heterogeneous collection in terms of formula properties. For this reason it turns out to be challenging to obtain statistically significant data that would admit drawing general conclusions.

The recent paper [EGCG⁺18] instead put forth the proposition that a better understanding of CDCL could be obtained by running experiments on carefully chosen theoretical benchmarks. By tuning various parameters, one can study what impact each heuristic has on performance and how this correlates with the theoretical properties of the formulas. An obvious objection is that it is very unclear why such a study of crafted benchmarks should have any practical relevance, but some arguments in favour of this approach given in [EGCG⁺18] are as follows:

- The benchmarks are *scalable*, meaning that one can generate “the same” formula for different sizes and study how performance scales as the instance size increases. (This simple but powerful idea was perhaps first articulated clearly in an applied SAT solving setting in [PV05].)
- The benchmarks are chosen to have different *extremal* properties in a proof-complexity-theoretic sense, meaning that they can be viewed as challenging benchmarks for different heuristics for variable decisions, clause deletions, restarts, et cetera.
- Finally, in contrast to most combinatorial benchmarks traditionally used in the *SAT competitions* [SAT], which are known to be very hard for resolution, the benchmarks in [EGCG⁺18] have been constructed so as to be *easy* in the sense of having very short resolution proofs of unsatisfiability that CDCL solvers could potentially find. In view of this, it can be argued that the performance of the solver provides a measure of the quality of the proof search and how it is affected by different heuristics.

Below follow the main conclusions reported in [EGCG⁺18] after comparing the empirical results with theoretical properties of the benchmarks:

1. Learned clauses are absolutely critical for performance. It is true that the information gathered while *learning* the clauses is important for guiding other heuristics, but this is not enough — the solvers crucially need to also *store* the clauses to realize the exponential increase in reasoning power from tree-like (DPLL-style) proofs to DAG-like resolution proofs.
2. While the mathematical question of whether restarts are just a helpful heuristic or are fundamentally needed for CDCL solvers to harness the full power of resolution remains wide open, the experimental results provide some circumstantial evidence that the latter might be the case. Also, *adaptive restarts* as in [AS12] often work markedly better than the fixed-interval so-called *Luby sequence* restarts in [ES04]. More support for this observa-

tion was provided very recently in [KN20].

3. For formulas inspired by time-space trade-off results, too aggressive clause erasure can incur a stiff penalty in running time also in practice. And when memory is tight, the *literal block distance (LBD)* heuristic [AS09] often does a particularly good job at identifying useful clauses.
4. For VSIDS variable decisions [MMZ⁺01] the choice of decay factor can sometimes be vitally important. It is not at all clear why, but one hypothesis is that this might be connected to whether the proof search needs to find DAG-like proofs or whether tree-like proofs are good enough. One can also see that VSIDS decisions can sometimes go badly wrong for easy but tricky formulas, which suggests that there is room for further improvements in variable selection heuristics. Somewhat disappointingly, [EGCG⁺18] did not find any support for the hypothesis that the newly proposed *learning-rate based branching (LRB)* heuristic [LGPC16] would be better than, or even distinguishable from, VSIDS.

It should perhaps be stressed that none of these findings should be considered to be a priori obvious, since proof complexity is inherently non-constructive whereas CDCL is about algorithmic proof search. Another point to emphasize is that the above findings are in no way rigorous results, but more a way of collecting circumstantial evidence for connections between theory and practice. Much work still remains to gain a more solid, mathematical understanding of when and why CDCL solvers work.

7.5. Algebraic Proof Systems

We now switch topics to *algebraic proof systems*, where formulas are translated to polynomials so that questions of satisfiability or unsatisfiability can be answered using algebraic methods of reasoning.

In what follows, we will let \mathbb{F} be a field (which will usually be the finite field $\text{GF}(2)$ with two elements $\{0, 1\}$ in practical SAT solving applications, but can be any field from the point of view of proof complexity) and $\vec{x} = \{x_1, \dots, x_n\}$ be a set of variables. Algebraic proof systems deal with polynomials in the polynomial ring over \mathbb{F} , i.e., multivariate polynomials in the variables x_1, \dots, x_n with coefficients from \mathbb{F} . A *monomial* m is a product of variables $m = \prod_{i=1}^n x_i^{e_i}$ for $e_i \in \mathbb{N}$. If $e_i \in \{0, 1\}$ for all i we say that the monomial is *multilinear* (this is sometimes also referred to as being *square-free*). A *term* $t = \alpha m$ is a monomial m multiplied by some field element $\alpha \in \mathbb{F}$.²⁸

7.5.1. Nullstellensatz

As our first example of an algebraic proof system we discuss *Nullstellensatz* introduced by Beame et al. [BIK⁺94]. A *Nullstellensatz refutation* of a set of polynomials $\mathcal{P} = \{p_i(\vec{x}) \mid i \in [m]\}$ in the polynomial ring $\mathbb{F}[\vec{x}]$ is a syntactic equality

²⁸We remark that the exact meaning of the terms “monomial” and “term” varies in the literature, but our choice of terminology here seems well in line with common usage in proof complexity.

$$\sum_{i=1}^m r_i(\vec{x}) \cdot p_i(\vec{x}) + \sum_{j=1}^n s_j(\vec{x}) \cdot (x_j^2 - x_j) = 1, \quad (7.15)$$

where r_i, s_j are also polynomials in $\mathbb{F}[\vec{x}]$. In algebraic language, what this shows is that the multiplicative identity 1 of the field \mathbb{F} lies in the polynomial ideal generated by $\mathcal{P} \cup \{x_j^2 - x_j \mid j \in [n]\}$. By Hilbert's Nullstellensatz, such a refutation (7.15) exists if and only if there is no common $\{0, 1\}$ -valued root for the set of polynomials \mathcal{P} .

Nullstellensatz can also be viewed as a proof system for certifying the unsatisfiability of CNF formulas. In this setting we first translate clauses of the form

$$C = \bigvee_{x \in P} x \vee \bigvee_{y \in N} \bar{y} \quad (7.16a)$$

to polynomials

$$p(C) = \prod_{x \in P} (1 - x) \cdot \prod_{y \in N} y. \quad (7.16b)$$

As a concrete example, $D = x \vee y \vee \bar{z}$ gets translated to $p(D) = (1 - x)(1 - y)z = z - yz - xz + xyz$.²⁹ Then a Nullstellensatz refutation of $\{p(C_i) \mid i \in [m]\}$ can be viewed as a refutation of the CNF formula $F = \bigwedge_{i=1}^m C_i$. This is so since an assignment to the variables in F (where we think of 1 as true and 0 as false) is satisfying precisely if all the polynomials $\{p(C_i) \mid i \in [m]\}$ vanish, i.e., evaluate to 0, and a Nullstellensatz refutation rules out that such a satisfying assignment exists.

The *size* of a Nullstellensatz refutation is defined to be the total number of monomials encountered when all products of polynomials are expanded out as linear combinations of monomials. To be more precise, let $mSize(p)$ denote the number of monomials in a polynomial p written as a linear combination of monomials (so that for our example clause $D = x \vee y \vee \bar{z}$ above we have $mSize(p(D)) = 4$). Then the size of a Nullstellensatz refutation of the form (7.15) is

$$\sum_{i=1}^m mSize(r_i(\vec{x})) \cdot mSize(p_i(\vec{x})) + \sum_{j=1}^n 2 \cdot mSize(s_j(\vec{x})). \quad (7.17)$$

We remark that this is not the only possible way of measuring size. It can be noted that the definition (7.17) is quite wasteful in that it forces us to represent the proof in a very inefficient way. Other papers in the so-called *semialgebraic* proof complexity literature, such as [GHP02b, KI06, DMR09], instead define size

²⁹Note that in this translation we are thinking of 1 as *true* and 0 as *false*, which is standard in proof complexity. It can be argued, though, that in the specific context of algebraic proof systems a more natural convention is to flip the values and identify 0 with *true* and 1 with *false*, just as a clause evaluating to true is identified with the corresponding polynomial evaluating to 0. If we adopt this flipped convention, then $x \vee y \vee \bar{z}$ would be translated to $xy(1 - z) = xy - xyz$. There is no clear consensus on this matter in the algebraic proof complexity literature, however, and so for simplicity we will identify 0 with *false* and 1 with *true* throughout this survey to be consistent with how other proof systems view the constants 0 and 1.

in terms of the polynomials in isolation, more along the lines of

$$\sum_{i=1}^m (m\text{Size}(r_i(\vec{x})) + m\text{Size}(p_i(\vec{x}))) + \sum_{j=1}^n (m\text{Size}(s_j(\vec{x})) + 2) \quad (7.18)$$

or as the bit size or “any reasonable size” of the representation of all polynomials $r_i(\vec{x}), p_i(\vec{x}), s_j(\vec{x})$. However, our definition (7.17) is consistent with the general definition of size for so-called algebraic and semialgebraic proof systems in [ALN16, Ber18, AO19], and, in particular, matches the size definition in the other algebraic proof systems that will be discussed later in this section.³⁰

A much more well-studied measure for Nullstellensatz than size is *degree*, which we define as $\max\{\deg(r_i(\vec{x}) \cdot p_i(\vec{x})), \deg(s_j(\vec{x}) \cdot (x_j^2 - x_j))\}$. In order to prove a lower bound d on Nullstellensatz degree for refuting \mathcal{P} , one can construct a d -*design*, which is a map D from degree- d polynomials in $\mathbb{F}[\vec{x}]$ to \mathbb{F} such that

1. D is linear, i.e., $D(\alpha p + \beta q) = \alpha D(p) + \beta D(q)$ for $\alpha, \beta \in \mathbb{F}$;
2. $D(1) = 1$;
3. $D(rp) = 0$ for all $p \in \mathcal{P}$ and $r \in \mathbb{F}[\vec{x}]$ such that $\deg(rp) \leq d$;
4. $D(x^2s) = D(xs)$ for all $s \in \mathbb{F}[\vec{x}]$ such that $\deg(s) \leq d - 2$.

Designs provide a characterization of Nullstellensatz degree in that there is a d -design for \mathcal{P} if and only if there is no Nullstellensatz refutation of \mathcal{P} in degree d (this is clearly spelled out in [Bus98] but is mentioned there to have been known before). The only-if direction is clear — applying D to a purported Nullstellensatz refutation of the form (7.15) yields 0 on the left-hand side but 1 on the right-hand side, which is a contradiction. The if-direction requires more work, but follows from linear programming duality.

Lower bounds on Nullstellensatz degree have been proven for formulas encoding combinatorial principles such as the pigeonhole principle [BCE⁺98] and pebbling contradictions [BOCIP02], which have already been described in previous sections, and also for other formulas encoding the *induction principle* [BP98b], *house-sitting principle* [CEI96, Bus98], and *matching* [BIK⁺97], most of which we will not discuss further in this survey.

It seems fair to say that research in algebraic proof complexity soon moved on from Nullstellensatz to stronger systems such as *polynomial calculus* [CEI96], which we will discuss shortly. Briefly (and somewhat informally), the difference between Nullstellensatz and polynomial calculus is that in the latter proof system the proof that 1 lies in the ideal generated by $\mathcal{P} \cup \{x_j^2 - x_j \mid j \in [n]\}$ can be constructed dynamically by a step-by-step derivation, which sometimes makes it possible to decrease both degree and size significantly (while, in the opposite direction, the many lower bounds on degree and size later established for

³⁰We remark that in the end the difference between these two theoretical size measures is not too important, since the two measures (7.17) and (7.18) are at most a square apart, and when measuring size in proof complexity we typically focus on the distinction between polynomial and superpolynomial. In addition, and more importantly, when we are dealing with k -CNF formulas with $k = O(1)$, as we are mostly doing in this survey, then the two size definitions are the same up to a constant factor 2^k . We refer the reader to Section 2.4 in [AH19] for a more detailed discussion of the definition of proof size in algebraic and semialgebraic proof systems.

polynomial calculus, as discussed in Section 7.5.2, certainly apply also to Nullstellensatz). The Nullstellensatz proof system has seen somewhat of a revival in a recent line of works [RPRC16, PR17, PR18, GKRS19, dRMN⁺20] showing that Nullstellensatz degree lower bounds can be “lifted” to lower bounds in stronger computational models. We will briefly discuss lifting in Section 7.7.1 and then see some examples of these ideas can be used in proof complexity in the rest of Section 7.7. The size complexity measure for Nullstellensatz has also received attention in recent papers such as [Ber18, AO19, dRNM19].

When proving lower bounds for algebraic proof systems it is often convenient to consider a *multilinear* setting where refutations are presented in the quotient ring $\mathbb{F}[\vec{x}]/\{x_j^2 - x_j \mid j \in [n]\}$. Since the Boolean axioms $x_j^2 - x_j$ are no longer needed, the refutation (7.15) can be written simply as

$$\sum_{i=1}^m r_i(\vec{x}) \cdot p_i(\vec{x}) = 1, \quad (7.19)$$

where we assume that all results of multiplications are implicitly multilinearized. It is clear that any refutation on the form (7.15) remains valid after multilinearization, and so the size and degree measures can only decrease in a multilinear setting.

7.5.2. Polynomial Calculus

The proof system *polynomial calculus* (*PC*) was introduced in [CEI96] to model Gröbner basis computations (and was originally called the “Gröbner basis proof system,” although by now the name “polynomial calculus” is firmly established). As in Nullstellensatz, the setup is that we have a set of polynomials \mathcal{P} over variables x_1, \dots, x_n , where the polynomial coefficients live in some field \mathbb{F} . The goal is to prove that the polynomials $\mathcal{P} \cup \{x_j^2 - x_j \mid j \in [n]\}$ have no common root. When operating with CNF formulas, we first translate the clauses to polynomials using the transformation from (7.16a) to (7.16b).

It is important to observe that, from an algebraic point of view, the variables can take as values any elements in the field \mathbb{F} . Hence, we need to add constraints enforcing 0/1 assignments. We are also allowed to take linear combinations of polynomials, or to multiply a polynomial by any monomial, since any common root for the original polynomials is preserved under such operations. This leads to the following set of derivation rules for polynomial calculus:

$$\text{Boolean axioms} \quad \frac{}{x_j^2 - x_j} \quad (7.20a)$$

$$\text{Linear combination} \quad \frac{p}{\alpha p + \beta q} \quad (\alpha, \beta \in \mathbb{F}) \quad (7.20b)$$

$$\text{Multiplication} \quad \frac{p}{mp} \quad (m \text{ any monomial}) \quad (7.20c)$$

A polynomial calculus refutation of \mathcal{P} also allows polynomials from \mathcal{P} as axioms; the refutation ends when 1 has been derived, showing that the polynomial equations have no common root, or equivalently when \mathcal{P} is an translation of a CNF

formula, that this formula is unsatisfiable. The polynomial calculus proof system is sound and complete, not only for CNF formulas but also for inconsistent systems of polynomial equations in general. As for Nullstellensatz, we can consider the setting where all polynomials are multilinear, because any higher powers of variables can always be eliminated using the Boolean axioms (7.20a).

To define the complexity measures of *size*, *degree*, and *space*, we write out the polynomials in a refutation as linear combinations of monomials. Then the *size* of a refutation, which is the analogue of resolution length, is the total number of monomials in the refutation (counted with repetitions), the *degree*, corresponding to resolution width, is the largest degree of any monomial in it, and the (*monomial*) *space*, which is the analogue of resolution (clause) space, is the maximal number of monomials in memory at any point during the refutation (again counted with repetitions). One can also define a *length* measure for polynomial calculus, which is the number of derivation steps, but this can be exponentially smaller than the size, which is the more relevant measure to study here.³¹

The representation of a clause $\bigvee_{i=1}^n x_i$ as a polynomial in polynomial calculus is $\prod_{i=1}^n (1 - x_i)$, which means that the number of monomials is exponential in the clause width. This problem arises only for positive literals, however — a large clause with only negative literals is translated to a single monomial. To get a cleaner and more symmetric treatment, in [ABSRW02] the proof system *polynomial calculus (with) resolution*, or *PCR* for short, was introduced. The idea is to add an extra set of parallel formal variables x'_j , $j \in [n]$, sometimes referred to as *twin variables*, so that positive and negative literals can both be represented in an efficient fashion.

Lines in a PCR proof are polynomials over the ring $\mathbb{F}[x_1, x'_1, \dots, x_n, x'_n]$, where as before \mathbb{F} is some field. We have all the axioms and rules of polynomial calculus plus the axiom

$$\text{Negation} \quad \frac{}{x_j + x'_j - 1} . \quad (7.20d)$$

Size, length, and degree are defined as for polynomial calculus, and the (monomial) space of a PCR refutation is again the maximal number of monomials in any configuration counted with repetitions.

It is important to understand that from an algebraic point of view the variables x_j and x'_j are completely independent. The point of the negation rule, therefore, is to force x_j and x'_j to take opposite values in $\{0, 1\}$, so that they respect the intended meaning of negation. It is worth noting that in actual Gröbner basis calculations one would not have both x_j and x'_j , so the introduction of “twin variables” is just to get a nicer proof system from a theoretical point of view. Our example clause $D = x \vee y \vee \bar{z}$ is rendered as $x'y'z$ in PCR.

One gets the same degree bounds for polynomial calculus resolution as in polynomial calculus, (just substitute $1 - x$ for x'), but one can potentially avoid an exponential blow-up in size measured in the number of monomials (and thus also for space). There are k -CNF formulas for which PCR is exponentially more

³¹In fact, if we consider the multilinear setting, where there are no Boolean axioms and instead multiplication is defined to return the multilinearized result, then it is not hard to show that any CNF formula with m clauses over n variables can be refuted in polynomial calculus in length $O(mn)$. See, e.g., [MN15] for a proof of this fact.

$$\begin{array}{c}
 \frac{x \vee \bar{y} \vee z \quad \bar{y} \vee \bar{z}}{x \vee \bar{y}} \\
 \text{(a) Resolution step.}
 \end{array}
 \qquad
 \begin{array}{c}
 \frac{\frac{\mathbf{yz}}{x'yz} \quad \frac{z' + z - 1}{x'yz' + x'yz - x'y}}{\mathbf{x'yz'}}}{\mathbf{x'y}} \\
 \text{(b) Corresponding PCR derivation.}
 \end{array}$$

Figure 7.13. Example of simulation of resolution step by PCR.

powerful than PC with respect to size [dRLM⁺20]. In PCR, monomial space is a natural generalization of clause space since every clause translates into a monomial as just explained in the example above.

Clearly, PC and PCR are very closely related, and in what follows we will sometimes be a bit sloppy and write just “polynomial calculus” when the distinction between the two is not important. We write “polynomial calculus resolution” or “PCR” to highlight when a claim only holds for polynomial calculus with twin variables for positive and negative literals.

7.5.3. Nullstellensatz, Polynomial Calculus, and Resolution

Polynomial calculus resolution can simulate resolution efficiently with respect to length/size, width/degree, and space simultaneously simply by mimicking refutations step by step. This means that all worst-case upper bounds for resolution immediately carry over to PCR. For an example of how this works, see the simulation of the resolution step in Figure 7.13a by the derivation in Figure 7.13b, where the polynomials corresponding to the simulated clauses are in boldface.

Polynomial calculus can be strictly stronger than resolution with respect to size and degree. For instance, over $\text{GF}(2)$ it is not hard to see that Tseitin formulas can be refuted in size $O(N \log N)$ and degree $O(1)$ by doing Gaussian elimination. Another example are the onto functional pigeonhole principle (FPHP) formulas (7.7a)–(7.7d), which were shown to be easy in [Rii93]. It remains open whether such separations can be found also for space, however.

Open Problem 7.5. Prove (or disprove) that polynomial calculus resolution is strictly stronger than resolution with respect to space.³²

The proof systems Nullstellensatz and polynomial calculus without twin variables are *incomparable* to resolution with respect to size/length — there are formulas for which both Nullstellensatz and PC are exponentially more efficient than resolution, and other formulas for which resolution is exponentially better.

7.5.4. Size and Degree for Nullstellensatz and Polynomial Calculus

A lot of what is known about length versus width in resolution carries over to size versus degree in polynomial calculus, whereas Nullstellensatz is mostly different.

³²A constant-factor separation is claimed in [ABSRW02], but this result is for the slightly different monomial space measure where every distinct monomial is only charged once but an arbitrary number of copies of this monomial (in different polynomials) can be had for free.

It is not hard to show that for both Nullstellensatz and polynomial calculus upper bounds on degree imply upper bounds on size, in the sense that if a CNF formula over n variables can be refuted in degree d , then such a refutation can be carried out in size $n^{O(d)}$. This is qualitatively similar to the bound for resolution, although the arguments are a bit more involved. Just as for resolution, this upper bound has been proven to be tight up to constant factors in the exponent for polynomial calculus [ALN16], and it follows from [DLLMO09a] that this also holds for Nullstellensatz.

In the other direction, a lower bound on size in terms of degree exactly analogous to the size-width bound (7.9) for resolution [BSW01] holds also for polynomial calculus, as shown in [IPS99]. For Nullstellensatz it is not possible to obtain lower bounds on size from degree lower bounds in this way, and pebbling formulas provide a counter-example [BOCIP02].

Interestingly, the paper [IPS99] is a precursor to [BSW01], and although it was far from obvious at the time it turns out that one can run exactly the same proof for both resolution and polynomial calculus. As for resolution, the ordering principle formulas in (7.11a)–(7.11d) witness the optimality of this size-degree lower bound, as shown by [GL10b]. As for resolution, almost all size lower bounds are derived via degree lower bounds.

The basic tool for proving polynomial calculus degree lower bounds is that of R -operators, which are analogues of the d -designs used for Nullstellensatz. As proposed in [Raz98], the idea is to give an overapproximation of what polynomials can be derived in degree at most d by defining an operator R on multilinear polynomials such that all degree- d consequences of the axioms are contained in the set $\{p \mid R(p) = 0\}$. The degree lower bound then follows by showing that $R(1) \neq 0$.

Formally, let $d \in \mathbb{N}^+$ be a positive integer. Suppose that there exists a linear operator R on a set \mathcal{P} of (multilinear) polynomials of degree at most d with the following properties:

1. $R(1) \neq 0$.
2. $R(f) = 0$ for all axioms $f \in \mathcal{P}$.
3. For every term t with $\text{Deg}(t) < d$ and every variable x it holds that $R(xt) = R(xR(t))$.

Then any polynomial calculus refutation of \mathcal{P} requires degree strictly greater than d . (Note that here we can restrict our attention to PC without twin variables for literals, since the degree measure is the same for PC and PCR.)

The proof of this claim is not hard. The basic idea is that R will map all axioms to 0 by property 2, and further derivation steps in degree at most d will yield polynomials that also map to 0 by the linearity of R and property 3 (where we use that without loss of generality we can implement multiplication by a monomial by multiplying by all variables in it one by one). But then property 1 implies that no derivation in degree at most d can reach 1 and establish contradiction. However, constructing such operators to obtain degree lower bounds seems much harder than proving resolution width lower bounds, and the technical machinery is much less well developed.

With the exception of Tseitin formulas and onto functional pigeonhole prin-

ciple (FPHP) formulas, all the formulas discussed in detail in Section 7.4.1 are equally hard also with respect to polynomial calculus size, which can be shown via degree lower bounds arguments:

- Hardness of the standard CNF encoding (7.7a)–(7.7b) of pigeonhole principle formulas³³ follows from [AR03], with some earlier work on other non-CNF encodings in [Raz98, IPS99]. The proof in [AR03] works also if onto clauses (7.7d) are added, and more recently it was shown in [MN15] that FPHP formulas with clauses (7.7a)–(7.7c) are also hard (whereas with both onto and functionality axioms added the formulas are easy, as noted above).
- Strong degree and size lower bounds on random k -CNF formulas were shown in [BSI99] for polynomial calculus over fields of characteristic distinct from 2, and lower bounds in any characteristic including 2 were established by different methods in [AR03].
- For the subset cardinality formulas in Figure 7.10, polynomial calculus degree and size lower bounds were obtained in [MN14].
- Also, “Tseitin formulas with the wrong modulus” are hard — one can define Tseitin-like formulas encoding counting modulo primes q , and such formulas are hard over fields of characteristic $p \neq q$ [BGIP01, AR03].

We also wish to discuss briefly two formula families for which we currently do not know how to prove lower bounds on polynomial calculus degree or size. Recall that a *legal k -colouring* of an undirected graph $G = (V, E)$ is a mapping $\chi : V \rightarrow [k]$ such that for every edge $(u, v) \in E$ it holds that $\chi(u) \neq \chi(v)$. The *k -colouring formula* for a graph G is the set of polynomials

$$\sum_{j=1}^k x_{v,j} - 1 \qquad v \in V(G), \qquad (7.21a)$$

$$x_{v,j}x_{v,j'} \qquad v \in V(G), j \neq j' \in [k], \qquad (7.21b)$$

$$x_{u,j}x_{v,j} \qquad (u, v) \in E(G), j \in [k], \qquad (7.21c)$$

with the intended interpretation that $x_{v,j} = 1$ if vertex v has colour $\chi(v) = j$. It is clear that these polynomials have a common $\{0, 1\}$ -valued root if and only if there exists a legal k -colouring, in which case we say that G is *k -colourable*. We wish to prove lower bounds on the complexity of disproving this formula for graphs G that are *not* k -colourable. (We remark that we can also represent (7.21a)–(7.21c) as a CNF formula by rewriting the only non-clausal constraints (7.21a) as the clause $\bigvee_{j=1}^k x_{v,j}$, but the exact representation does not matter for this discussion.)

For resolution, it was shown in [BCMM05] that if one samples a graph G at random with appropriate edge probabilities, then with overwhelming probability G is not k -colourable, but resolution still requires linear width, and hence exponential length, to refute the k -colouring formula over G .³⁴ This gives a very strong

³³Here a twist is needed since these formulas have high initial degree, but we will not go into this. The most elegant solution is to consider so-called graph PHP formulas as discussed in, e.g., [BSW01, MN15].

³⁴CNF versions (in the standard DIMACS format used by SAT solvers) of 3-colouring formulas over n -vertex random graphs with appropriate parameters can be obtained with

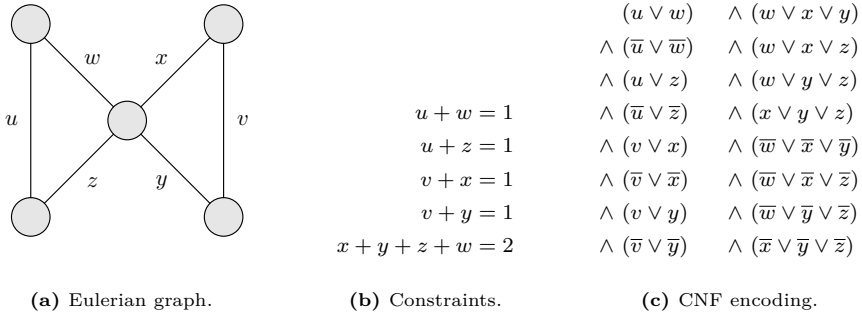


Figure 7.14. Example of even colouring (EC) formula (satisfiable instance).

average-case lower bound over a whole distribution of inputs. For polynomial calculus, all that is known is that one can construct specific worst-case instances that are exponentially hard [LN17], but we cannot yet rule out that k -colouring would be an easy problem for Gröbner basis computations in an average-case setting (although it seems absurd that this would be the case).

Open Problem 7.6. Prove linear lower bound on polynomial calculus degree, and hence exponential lower bounds on size, for k -colouring formulas over randomly sampled graphs analogously to the average-case lower bound for resolution in [BCMM05].

Another quite intriguing family of benchmark formulas are the *even colouring (EC) formulas* constructed by Markström [Mar06] (see Figure 7.14 for an example). These formulas are defined over connected graphs with all vertices having even degree. For a fixed graph $G = (V, E)$ we consider every edge $e \in E$ to be a variable and let the formula consist of the (CNF encodings of) the constraints

$$\sum_{e \in E(v)} e = \deg(v)/2 \quad v \in V, \quad (7.22)$$

where $E(v)$ denotes the set of edges incident to v . The constraints (7.22) require that the edges should be labelled 0/1 in such a way that for every vertex v in V the number of 0-edges and 1-edges incident to v is equal. If the total number of edges in the graph is even, then this formula is satisfiable — since the graph has all vertices of even degree it also has an Eulerian cycle, and we can fix any such cycle and label every second edge 0 and 1, respectively. If the number of edges is odd, however, then summing up (7.22) over all vertices yields $2 \cdot \sum_{e \in E(G)} e = |E(G)|$ which is clearly a contradiction since an odd number cannot be divisible by 2.

the tool *CNFgen* [LENV17, CNF] using the command line `cnfgen kcolor 3 gnp <n> <p>` for $p = 5/(n - 1)$. These formulas start getting quite challenging for SAT solvers around $n = 800$. (It should be noted that this edge density is slightly too low to get the theoretical guarantees from [AM99] that the graphs are not 3-colourable, and thus that the formulas are unsatisfiable, asymptotically almost surely, but empirically this seems to hold. Choosing $p = 5.5/(n - 1)$ should be enough to also get theoretical guarantees, but yields more overconstrained formulas, meaning that n has to be chosen larger before the exponential hardness manifests itself.)

If the graph G is a good enough expander graph, then a fairly standard extension of the techniques in [BSW01] can be used to prove that the even colouring formula over G requires linear width, and hence also exponential length, for resolution.³⁵ However, a moment of reflection reveals that over $\text{GF}(2)$ it is easy to derive polynomials $\sum_{e \in E(v)} e - \deg(v)/2$ from (7.22), and summing these polynomials for all $v \in V$ yields $2 \cdot \sum_{e \in E(G)} e - |E(G)| = 1$ (where the calculations are in $\text{GF}(2)$). Thus, even colouring formulas are easy over $\text{GF}(2)$, or more generally over any field of characteristic 2.

Open Problem 7.7. Are even colouring formulas over expander graphs hard for polynomial calculus over fields of characteristic distinct from 2?

7.5.5. Polynomial Calculus Space

We next turn to a discussion of space complexity. This measure does not make too much sense for Nullstellensatz, since refutations of the form (7.15) are presented in “one shot” and it is hard to talk about any intermediate results. Thus, our discussion of space will focus on polynomial calculus.

Recall that for resolution we measure space as the number of clauses in memory, and since clauses turn into monomials in polynomial calculus resolution the natural analogue here is monomial space (in our discussion of space we are always focusing on PCR, since this proof system allows for a more efficient representation of disjunctive clauses as discussed above). The first monomial space lower bounds were shown for pigeonhole principle formulas in [ABSRW02]. These formulas have wide axioms, however, and if one applies the 3-CNF conversion from (7.1a) to (7.1b) the lower bound proof breaks down.

Monomial space lower bounds for formulas of bounded width were proven only in [FLN⁺15] for an obfuscated 4-CNF version of PHP formulas. This was followed by optimal, linear lower bounds for random 4-CNF formulas [BG15], and then for Tseitin formulas over expanders but with the added assumptions that either these graphs are sampled randomly or one adds two copies of every edge to get a multigraph [FLM⁺13].³⁶ Somewhat intriguingly, none of these papers could show any nontrivial lower bounds for any 3-CNF formulas. This barrier was finally overcome by [BBG⁺17], where optimal lower bounds for random 3-CNF formulas were established. However, the following open problems show that we still do not understand polynomial calculus space very well.

Open Problem 7.8. Prove linear lower bounds on polynomial calculus space for refutations of Tseitin formulas over d -regular expander graphs for $d = 3$ or even $d > 3$ using no other assumptions than expansion only.

Open Problem 7.9. Prove that pigeonhole principle formulas require large monomial space in polynomial calculus even when converted to 3-CNF.

³⁵To generate unsatisfiable even colouring formulas over 6-regular random graphs, which are expanding asymptotically almost surely, one can use *CNFgen* with the command line `cnfgen ec gnd <n> 6` for the number of vertices n in the graph chosen odd. For CDCL solvers, these formulas become noticeably hard around $n = 21$.

³⁶It is worth noting that these space lower bounds hold for any characteristic, so although Tseitin formulas have small-size refutations over $\text{GF}(2)$, such refutations still require large space.

Another intriguing question is whether an analogue of the fact that resolution width is a lower bound for resolution space (7.12) holds also for polynomial calculus.

Open Problem 7.10. Is it true that $\text{space} \geq \text{degree} - O(1)$ for k -CNF formulas in polynomial calculus?

For a long time, essentially nothing was known about this problem, except that the work [FLM⁺13] made what can be described as some limited progress by showing that if a formula requires large resolution width (which is a necessary, but not sufficient, condition for high degree), then the XOR-substituted version of the formula (as in (7.13a)–(7.13c)) requires large monomial space. When applied to Tseitin-like formulas over expander graphs, this result yields an optimal separation of space and degree. Namely, it follows that these formulas can be refuted in degree $O(1)$ but require space $\Omega(N)$. To obtain such separations we have to commit to a finite characteristic p of the underlying field, however, and the formulas encoding counting mod p will separate space and degree only for fields of this characteristic. It would be nice to get a separation that would work in any characteristic, and the candidate formulas to obtain such a result readily present themselves.

Open Problem 7.11. Prove (or disprove) that XOR-substituted pebbling formulas as in Figure 7.12 require monomial space lower-bounded by the pebbling space of the underlying DAG (which if true would yield an essentially optimal space-degree separation independent of the field characteristic).

Recently, some quite exciting news regarding Open Problem 7.10 was announced [GKT19], namely that the PCR monomial space of refuting a formula is lower-bounded by the square root of the resolution refutation width (which, as mentioned above, is stronger than the corresponding lower bound in terms of degree, since resolution width can be much larger than polynomial calculus degree). This also implies lower bounds for the formulas mentioned in Open Problems 7.8 and 7.9 and provides a separation between degree and space that is independent of the characteristic as called for in Open Problem 7.11, although all of these results are off by a square root from what would have been expected. It is not clear whether the bounds obtained in this way are tight or not.

7.5.6. Trade-off Results for Nullstellensatz and Polynomial Calculus

When it comes to trade-offs in polynomial calculus we again recognize most of the picture from resolution, but there are also some differences. Here is a summary of what is known (where the upper bounds in all of the results hold for PC, i.e., polynomial calculus without twin variables, whereas the lower bound apply also for the PCR proof system with twin variables):

- For space versus degree in polynomial calculus we know strong, essentially optimal trade-offs from [BNT13], and the formulas exhibiting such trade-offs are the same vanilla pebbling contradictions as for resolution (for which we get exactly the same bounds as in Section 7.4.4). However, it is *not* known whether a result analogous to [BN20] holds, i.e., whether polynomial calculus refutations in small degree can require superlinear space.

- The paper [BNT13] also showed strong size-space trade-offs, and again the formulas used are substituted pebbling contradictions over appropriate DAGs and Tseitin formulas over long, narrow grids. Here there is some loss in parameters as compared to resolution, however, which seems to be due to limitations of the proof techniques rather than to actual differences in formula behaviour. Also, the Tseitin formula trade-off results do not hold over fields of characteristic 2. Overall, though, the size-space trade-offs are very similar to the length-space trade-offs for resolution discussed in Section 7.4.4, and we omit the details.
- The size blow-up in [IPS99] when degree is decreased is necessary, and can be obtained for k -CNF formulas of constant width $k = O(1)$, by using the same ideas as in [Tha16] and then adding a number of extra tweaks to make the argument go through in polynomial calculus as shown in [LNSS20].

In view of the above, we have two final open problems about polynomial calculus that we want to highlight in this section.

Open Problem 7.12. Establish size-space trade-offs for polynomial calculus that hold regardless of the characteristic of the underlying field (in contrast to [BNT13]).

Open Problem 7.13. Are there formulas over n variables for which polynomial calculus refutations in degree d require monomial space almost n^d , or at least $\omega(n)$?

We wish to mention also that for the weaker Nullstellensatz proof systems size-degree trade-offs were recently shown in [dRNMR19]. For instance, there is a family of 3-CNF formulas $\{F_n\}_{n=1}^{\infty}$ of size $\Theta(n)$ such that:

1. There is a Nullstellensatz refutation of F_n in degree $O(\sqrt[6]{n} \log n)$.
2. There is a Nullstellensatz refutation of F_n of nearly linear size $O(n^{1+\epsilon})$ and degree $O(\sqrt[3]{n} \log n)$.
3. Any Nullstellensatz refutation of F_n in degree at most $\sqrt[3]{n}$ must have exponential size.

The formulas F_n are vanilla pebbling contradictions without substitution generated from suitably chosen graphs.

7.5.7. Algebraic SAT Solving

We conclude this section with a(n all too) brief discussion of algebraic SAT solving. There seems to have been quite some excitement, at least in the theory community, about the Gröbner basis approach to SAT solving after the paper [CEI96] appeared. However, the hoped for improvement in performance from this method failed to materialize in practice. Instead, the CDCL revolution happened. . .

Some Gröbner basis SAT solvers have been developed, the most notable example perhaps being *PolyBoRi* [BD09, BDG⁺09], but these solvers do not seem competitive with resolution-based solvers (and, sadly, *PolyBoRi* is no longer maintained). Some SAT solvers such as *March* [HvM06] and *CryptoMiniSat* [Cry] successfully implement Gaussian elimination [HvM05], but this is only very limited form of polynomial calculus reasoning.

Is it harder to build good algebraic SAT solvers than CDCL solvers? Or is it just that too little work has been done? (Witness that it took over 40 years for resolution-based SAT solvers to become really efficient.) Or is it perhaps a little bit of both? It can be noted in this context that it was recently shown that efficient proof search is NP-hard for Nullstellensatz and polynomial calculus [dRGN⁺20] (strengthening an earlier paper [GL10a]) — i.e., these proof systems are not automatable — but it is not clear that this should be an obstacle in practice, since very successful SAT solvers have been built on top of resolution despite the non-automatability of this proof system [AM20].

Whatever the answer is to these questions, it seems clear that one needs to find some kind of shortcut in order to use Gröbner bases for efficient SAT solving. A full Gröbner basis computation does too much work, since it allows us not only to decide satisfiability but also to read off the number of satisfying assignments, which is widely believed to be a strictly harder problem.

This slightly downbeat discussion of algebraic SAT solving should not be taken to mean that algebraic methods cannot be used for successfully solving hard combinatorial problems, however. To give a positive example, in the sequence of papers [DLLMM08, DLLMO9b, DLLMM11], a body of work that was recognised with the *INFORMS Computing Society Prize 2010*, the authors solve graph colouring problems (with great success) essentially by constructing Nullstellensatz certificates of non-colourability. Hence, for some NP-complete problems it seems that even lowly Nullstellensatz can be a quite powerful approach.

Another very interesting line of work is exemplified by the papers [RBK17, RBK18, KBK19] using Gröbner bases computations to attack the challenging problem of verifying multiplier circuits. As a part of this work, the authors develop a formal proof logging system to certify correctness, called *practical algebraic calculus (PAC)*, and this proof system is nothing other than polynomial calculus (but with the field \mathbb{F} chosen to be the rational numbers \mathbb{Q} rather than a finite field).

7.6. Cutting Planes and Pseudo-Boolean Solving

The *cutting planes* proof system [CCT87], which formalizes the integer linear programming algorithm in [Gom63, Chv73] and underlies so-called *pseudo-Boolean (PB) solvers*, operates with linear inequalities over the reals with integer coefficients. To reason about CNF formulas in cutting planes, the disjunctive clauses are translated to linear inequalities, which are then manipulated to derive a contradiction. Thus, the question of Boolean satisfiability is reduced to the geometry of polytopes over the real numbers. Just as algebraic proof systems can deal not only with translated CNF formulas but with arbitrary sets of polynomials, cutting planes can operate on arbitrary 0-1 integer linear constraints, which we will also refer to as *pseudo-Boolean constraints*. A *pseudo-Boolean formula* is a set of pseudo-Boolean constraints (also known as a 0-1 *integer linear program*).

In the standard proof complexity setting, we use only positive literals (un-negated variables) and identify \bar{z} with $1 - z$ so that, for instance, the clause $x \vee y \vee \bar{z}$ gets translated to $x + y + (1 - z) \geq 1$, or $x + y - z \geq 0$ after we have moved all integer constants to the right-hand side. However, in order to give a description of

cutting planes that is helpful also when we want to reason about pseudo-Boolean solvers, and in order to get compact notation, it is more useful to keep negated literals as variables in their own right, and to insist that all inequalities consist of linear combinations of (positive or negative) literals with only non-negative coefficients.³⁷ It will also be convenient here to use the notation x^σ , $\sigma \in \{0, 1\}$, mentioned in Section 7.2, where we recall that $x^1 = x$ and $x^0 = \bar{x}$. With this notation, assuming that our set of variables is $\{x_1, \dots, x_n\}$ we can write all linear constraints in *normalized form*

$$\sum_{i \in [n], \sigma \in \{0, 1\}} a_i^\sigma x_i^\sigma \geq A, \quad (7.23)$$

where for all $i \in [n]$ and $\sigma \in \{0, 1\}$ it holds that $a_i^\sigma \in \mathbb{N}$ and $\min\{a_i^0, a_i^1\} = 0$ (the latter condition specifies that variables occur only with one sign in any given inequality), and where $A \in \mathbb{N}^+$ (this constant term is often referred to as the *degree of falsity*, or just *degree*, in an applied pseudo-Boolean solving context). In what follows, all expressions of the type (7.23) are supposed to be in normalized form, and all sums are assumed to be taken over all combinations of $i \in [n]$ and $\sigma \in \{0, 1\}$ except as specified under the summation sign.

If the input is a CNF formula F we just view every clause $C \in F$ of the form

$$C = x_1^{\sigma_1} \vee x_2^{\sigma_2} \vee \dots \vee x_w^{\sigma_w} \quad (7.24a)$$

as a linear constraint

$$x_1^{\sigma_1} + x_2^{\sigma_2} + \dots + x_w^{\sigma_w} \geq 1. \quad (7.24b)$$

That is, a disjunctive clause is simply a constraint on the form (7.23) where $a_i^\sigma \in \{0, 1\}$ and $A = 1$ (in particular, our example clause $x \vee y \vee \bar{z}$ now becomes $x^1 + y^1 + z^0 \geq 1$). Hence, CNF formulas can be viewed as a special case of pseudo-Boolean formulas).

Pseudo-Boolean constraints can be exponentially more concise than CNF, as is shown by a comparison of the constraint

$$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 \geq 3 \quad (7.25a)$$

with

$$\begin{aligned} & (x_1 \vee x_2 \vee x_3 \vee x_4) \wedge (x_1 \vee x_2 \vee x_3 \vee x_5) \wedge (x_1 \vee x_2 \vee x_3 \vee x_6) \\ & \wedge (x_1 \vee x_2 \vee x_4 \vee x_5) \wedge (x_1 \vee x_2 \vee x_4 \vee x_6) \wedge (x_1 \vee x_2 \vee x_5 \vee x_6) \\ & \wedge (x_1 \vee x_3 \vee x_4 \vee x_5) \wedge (x_1 \vee x_3 \vee x_4 \vee x_6) \wedge (x_1 \vee x_3 \vee x_5 \vee x_6) \\ & \wedge (x_1 \vee x_4 \vee x_5 \vee x_6) \wedge (x_2 \vee x_3 \vee x_4 \vee x_5) \wedge (x_2 \vee x_3 \vee x_4 \vee x_6) \\ & \wedge (x_2 \vee x_3 \vee x_5 \vee x_6) \wedge (x_2 \vee x_4 \vee x_5 \vee x_6) \wedge (x_3 \vee x_4 \vee x_5 \vee x_6) \end{aligned} \quad (7.25b)$$

(note that we are assuming here that it is not allowed to introduce new variables to encode the constraints — as discussed in Section 7.8, such extension variables

³⁷This is analogous to how twin variables were introduced for polynomial calculus in Section 7.5.2, except that for cutting planes this does not affect the reasoning power of the proof system in any significant way.

can change the properties of proof systems dramatically). Constraints of the form

$$x_1^{\sigma_1} + x_2^{\sigma_2} + \dots + x_w^{\sigma_w} \geq A, \tag{7.26}$$

i.e., such that $a_i^\sigma \in \{0,1\}$ holds for all coefficients a_i^σ , are called *cardinality constraints*, since they encode that at least A of the literals in the constraint are true. We can also have general pseudo-Boolean constraints such as, say, $x_1 + 2\bar{x}_2 + 3x_3 + 4\bar{x}_4 + 5x_5 \geq 7$.

7.6.1. Pseudo-Boolean Rules of Reasoning

Not only are pseudo-Boolean constraints much more concise than clauses, but the rules used to manipulate them are also more powerful. Using the normalized form (7.23), the derivation rules in the standard proof complexity definition of cutting planes are

$$\text{Literal axioms} \frac{}{x_i^\sigma \geq 0} \tag{7.27a}$$

$$\text{Multiplication} \frac{\sum a_i^\sigma x_i^\sigma \geq A}{\sum ca_i^\sigma x_i^\sigma \geq cA} \quad c \in \mathbb{N}^+ \tag{7.27b}$$

$$\text{Addition} \frac{\sum a_i^\sigma x_i^\sigma \geq A \quad \sum b_i^\sigma x_i^\sigma \geq B}{\sum (a_i^\sigma + b_i^\sigma) x_i^\sigma \geq A + B} \tag{7.27c}$$

$$\text{Division} \frac{\sum ca_i^\sigma x_i^\sigma \geq A}{\sum a_i^\sigma x_i^\sigma \geq \lceil A/c \rceil} \quad c \in \mathbb{N}^+ \tag{7.27d}$$

where in the addition rule (7.27c) we implicitly assume that the result is rewritten in normalized form. Let us illustrate this by a small example. It is important to note that when we add literals of opposite sign, the result is

$$x_i^1 + x_i^0 = x_i^1 + (1 - x_i^1) = 1 \tag{7.28}$$

(which is just another way of saying that it will always be the case that exactly one of the literals x_i and \bar{x}_i is true). If we have the two constraints

$$x + 2y + 3z + 4\bar{w} \geq 5 \tag{7.29}$$

and

$$3x + 2\bar{y} + \bar{z} + \bar{w} \geq 3, \tag{7.30}$$

then by applying the addition rule (7.27c) we get the expression

$$(1 + 3)x + (2 - 2)y + (3 - 1)z + (4 + 1)\bar{w} \geq 5 + 3 - (2 + 1) \tag{7.31a}$$

which in the normalized form (7.23) becomes

$$4x + 2z + 5\bar{w} \geq 5 \tag{7.31b}$$

(where we suppress terms with zero coefficients). We note that when adding (7.29) and (7.30) to obtain (7.31b) the coefficients for y and \bar{y} cancel so that

this variable disappears. In general, when we add two constraints $\sum a_i^\sigma x_i^\sigma \geq A$ and $\sum b_i^\sigma x_i^\sigma \geq B$ such that there is a variable x_i and a $\sigma \in \{0, 1\}$ for which $a_i^\sigma = b_i^{1-\sigma} > 0$, we say that this is an instance of *cancelling addition*.

More generally, when two linear constraints $\sum a_i^\sigma x_i^\sigma \geq A$ and $\sum b_i^\sigma x_i^\sigma \geq B$ share a variable x_j for which $a_j^\sigma > 0$ and $b_j^{1-\sigma} > 0$ hold, then we can multiply the constraints by the smallest numbers c_A and c_B such that $c_A a_j^\sigma = c_B b_j^{1-\sigma}$, and then apply cancelling addition. If we return to the constraints (7.29) and (7.30) but now focus on the variable z , then we can multiply the second constraint by 3 and then add to get

$$\frac{x + 2y + 3z + 4\bar{w} \geq 5 \quad \frac{3x + 2\bar{y} + \bar{z} + \bar{w} \geq 3}{9x + 6\bar{y} + 3\bar{z} + 3\bar{w} \geq 9}}{10x + 4\bar{y} + 7\bar{w} \geq 9} \quad (7.32)$$

More abstractly, writing $d = \gcd(a_j, b_j)$ for the greatest common divisor of a_j and b_j , so that the smallest numbers c_A and c_B such that $c_A a_j = c_B b_j$ are $c_A = b_j/d$ and $c_B = a_j/d$, the formal specification of this rule is

$$\frac{a_j x_j^\sigma + \sum_{i \neq j, \sigma} a_i^\sigma x_i^\sigma \geq A \quad b_j x_j^{1-\sigma} + \sum_{i \neq j, \sigma} b_i^\sigma x_i^\sigma \geq B}{\sum_{i \neq j, \sigma} ((b_j a_i^\sigma / d) + (a_j b_i^\sigma / d)) x_i^\sigma \geq b_j A / d + a_j B / d - a_j b_j / d} \quad (7.33)$$

(where as before we implicitly assume that the result of the linear combination is put into normalized form). Note that this can be viewed as a kind of generalization of the resolution rule (7.2) from disjunctive clauses to general linear constraints. We therefore refer to (7.33) as *generalized resolution* (or sometimes *cancelling linear combination*), and we say that the two constraints are *resolved over* x_j . This rule essentially goes back to Hooker [Hoo88, Hoo92] (although Hooker's definition is slightly different in that the cancelling addition has to be followed by a division step). It is worth noting, though, that one difference compared to the resolution rule for disjunctive clauses is that it may be possible to resolve the same pair of constraints over several different variables, which can yield different results (as in (7.31b) and (7.32) above).

Given a set of linear inequalities, one can show that there is no $\{0, 1\}$ -valued solution by using the cutting planes rules to derive the inequality $0 \geq 1$ from the given linear inequalities. It is clear that such a refutation can exist only if the instance is indeed unsatisfiable. The other direction also holds, but requires more work to establish [Chv73, CCT87]; for the special case of translations of CNF formulas, this follows from the cutting planes can simulate resolution as discussed in Section 7.7.2 below).

We want to highlight that in the division rule (7.27d) (which is also referred to as the *Chvátal-Gomory cut* rule) we can divide with the common factor c on the left and then *divide and round up* the constant term on the right to the closest integer, since we know that we are only interested in $\{0, 1\}$ -valued solutions. This division rule is where the power of cutting planes lies. (And, indeed, this is how it must be, since a moment of thought reveals that the other rules are sound also for real-valued variables, and so without the division rule we would not be able to distinguish sets of linear inequalities that have real-valued solutions but no $\{0, 1\}$ -valued solutions.)

It is not hard to see that we can modify the definitions slightly to obtain a more cleanly stated *general division* rule

$$\frac{\sum a_i^\sigma x_i^\sigma \geq A}{\sum \lceil a_i^\sigma / c \rceil x_i^\sigma \geq \lceil A/c \rceil} \quad c \in \mathbb{N}^+ \tag{7.34}$$

without changing anything essential, since this rule can easily be simulated by using rules (7.27a) and (7.27d). Therefore, although the standard definition of division in the proof complexity literature is as in (7.27d), without loss of generality we will use rule (7.34) (though it should be noted that the correctness of (7.34) hinges on the fact that we are using normalized form).

A small example just to illustrate how the rules can be combined is the derivation

$$\frac{\begin{array}{l} 6x + 2y + 3z \geq 5 \\ \frac{x + 2y + w \geq 1}{2x + 4y + 2w \geq 2} \text{ Multiplication by 2} \end{array}}{8x + 6y + 3z + 2w \geq 7} \text{ Addition} \tag{7.35}$$

$$\frac{8x + 6y + 3z + 2w \geq 7}{3x + 2y + z + w \geq 3} \text{ Division by 3}$$

where we note that in this case we do not lose any information in the final division step. This is not always true when using the general division rule (7.34) — for instance, a further division of $3x + 2y + z + w \geq 3$ by 3 would yield the clause $x + y + z + w \geq 1$, which is a strictly weaker constraint.

Pseudo-Boolean solvers such as *Sat4j* [LP10] do not implement the full set of cutting planes derivation rules as presented above, however. To describe how they work, we need to introduce two other rules, namely

$$\text{Weakening} \frac{\sum_{(i,\sigma)} a_i^\sigma x_i^\sigma \geq A}{\sum_{(i,\sigma), i \neq j} a_i^\sigma x_i^\sigma \geq A - (a_j^0 + a_j^1)} \tag{7.36a}$$

and

$$\text{Saturation} \frac{\sum_{(i,\sigma)} a_i^\sigma x_i^\sigma \geq A}{\sum_{(i,\sigma)} \min\{a_i^\sigma, A\} \cdot x_i^\sigma \geq A} . \tag{7.36b}$$

The weakening rule is merely a convenient shorthand for one application each of the rules (7.27a), (7.27b), and (7.27c). Just as division, saturation is a special rule in that it is sound only for integral solutions. A second small toy example

$$\frac{\begin{array}{l} 2x + y + z + \geq 2 \\ \frac{3\bar{x} + 2y + 2\bar{z} + u + w \geq 5}{3\bar{x} + 2y + u + w \geq 3} \text{ Weakening on } z \end{array}}{7y + 3z + 2u + 2w \geq 6} \text{ Resolution on } x \tag{7.37}$$

$$\frac{7y + 3z + 2u + 2w \geq 6}{6y + 3z + 2u + 2w \geq 6} \text{ Saturation}$$

shows how weakening, generalized resolution, and saturation can be combined. (Again, the statement of the saturation rule in (7.36b) is crucially using normalized form.)

In the proof complexity literature the focus has been on investigating the general cutting planes proof system with the derivation rules (7.27a)–(7.27c)

and (7.34). To understand the reasoning power of pseudo-Boolean solvers, however, it makes sense to study also other versions of the cutting planes proof system where (i) the saturation rule (7.36b) is used instead of the division rule (*cutting planes with saturation*), or (ii) all applications of (7.27b) and (7.27c) have to be combined into applications of the generalized resolution rule (7.33) (*cutting planes with resolution*), or (iii) both of these modifications are applied at the same time (*cutting planes with saturation and resolution*). We will return to these different versions of the cutting planes proof system in Section 7.7.4.

7.6.2. Conflict-Driven Pseudo-Boolean Solving

To explain why the different flavours of cutting planes just introduced are interesting from an applied point of view, we next turn to pseudo-Boolean solvers and how they can be used to determine the satisfiability of sets of pseudo-Boolean constraints, which we recall are referred to as pseudo-Boolean formulas.

One approach to solving pseudo-Boolean formulas is to convert them to CNF, either lazily by learning clauses from PB constraints during conflict analysis, as in *clasp* [GKKS09] and one of the version in the *Sat4j* library [LP10], or eagerly by re-encoding the whole formula to CNF (typically introducing new auxiliary variables, or extension variables) and running a CDCL solver as in, e.g., *MiniSat+* [ES06], *Open-WBO* [MML14], or *NaPS* [SN15]. In the context of cutting planes we are more interested in solvers doing native pseudo-Boolean reasoning, such as *PRS* [DG02], *Galena* [CK05], *Pueblo* [SS06], *Sat4j* [LP10], and *RoundingSat* [EN18], so this is where our focus will be below (we mention, though, that related, but slightly different, ideas were also explored in *bsolo* [MMS06]). Our discussion of pseudo-Boolean solvers cannot come anywhere close to doing full justice to the topic of pseudo-Boolean solving or the even richer area of pseudo-Boolean optimization — for this, we refer the reader to Chapter 28 in this handbook. Another source of much useful information is Dixon’s PhD thesis [Dix04].

In our discussion of pseudo-Boolean solving, the standard setting is that the input is a PB formula without 0-1 solutions, and the goal of the solver is to decide that the formula is contradictory. For readers more interested in optimization, note that the above scenario arises also when the optimal solution has been found and the solver should prove that the (linear) objective function cannot be better than in the current solution.

Just as CDCL solvers can be viewed as searching for resolution proofs, we will see that the pseudo-Boolean solving techniques we will discuss generate proofs in different subsystems of cutting planes. Simplifying drastically, when building a pseudo-Boolean solver on top of cutting planes we have the following choices:

1. Boolean rule: (a) saturation or (b) division.
2. Linear combinations: (a) generalized resolution or (b) no restrictions.

As we will soon see, the choice of generalized resolution seems inherent in a conflict-driven setting, which is what we are focusing on here. However, which Boolean rule to prefer is less clear. Saturation was used in the seminal paper [CK05] and has also been the rule of choice in what is arguably the most popular pseudo-Boolean solver to date, namely *Sat4j* [LP10]. Division appeared

only recently in *RoundingSat* [EN18] (although it was suggested in a more general integer linear programming setting in [JdM13]). We will return to a discussion of saturation versus division later, but let us first describe the general setup.

Naively, when generalizing CDCL to a pseudo-Boolean setting we just want to build a solver that decides on variable values and propagates forced values until conflict, at which point a new linear constraint is learned and the solver backtracks. To decide when constraints are propagating or conflicting it is convenient to use the concept of *slack*, which we define next.

The *slack* of a constraint $\sum_{i \in [n], \sigma \in \{0,1\}} a_i^\sigma x_i^\sigma \geq A$ under a partial assignment ρ (which we should think of as the variables currently assigned on the trail) measures how far $\sum_{i \in [n], \sigma \in \{0,1\}} a_i^\sigma x_i^\sigma \geq A$ is from being falsified by ρ , and is defined as

$$\text{slack} \left(\sum_{i \in [n], \sigma \in \{0,1\}} a_i^\sigma x_i^\sigma \geq A; \rho \right) = \sum_{\rho(x_i^\sigma) \neq 0} a_i^\sigma - A, \quad (7.38)$$

i.e., the sum of the coefficients of all literals that are *not* falsified by ρ minus the constant term. To illustrate with a special case, the slack of a disjunctive clause is the number of non-falsified literals in the clause minus 1. The constraint $\sum_{i \in [n], \sigma \in \{0,1\}} a_i^\sigma x_i^\sigma \geq A$ is *conflicting* under ρ if

$$\text{slack} \left(\sum_{i \in [n], \sigma \in \{0,1\}} a_i^\sigma x_i^\sigma \geq A; \rho \right) < 0 \quad (7.39)$$

and, for x_{i^*} not in the domain of ρ , it *propagates* $x_{i^*}^{\sigma^*}$ under ρ if

$$0 \leq \text{slack} \left(\sum_{i \in [n], \sigma \in \{0,1\}} a_i^\sigma x_i^\sigma \geq A; \rho \right) < a_{i^*}^{\sigma^*} \quad (7.40)$$

(which is just another way of saying that we have to set $x_{i^*}^{\sigma^*}$ to true, or else there will be no way to satisfy the constraint). The above definitions might be easier to digest by studying the example in Figure 7.15 of how the slack changes for the constraint $C \doteq x_1 + 2\bar{x}_2 + 3x_3 + 4\bar{x}_4 + 5x_5 \geq 7$ under different partial assignments ρ on the trail. The initial slack is just the sum of the coefficients minus the degree. If x_5 is assigned to false, then x_4 is propagated to false according to (7.40). Assigning a variable to its propagated value does not change the slack. If now the solver for some other reason sets x_3 to false and x_2 to true, we have a conflict according to (7.39). Note that in contrast to clauses in CDCL, a pseudo-Boolean constraint can be conflicting even though not all variables in it have been assigned.

7.6.2.1. A First (Failed) Attempt at Pseudo-Boolean Conflict Analysis

Now we can give a slightly more detailed (though still incomplete) sketch of how we would like our pseudo-Boolean solver to work:

1. While there is no conflict, iteratively propagate all literals $x_{i^*}^{\sigma^*}$ such that there is a constraint for which $0 \leq \text{slack}(\sum_{i \in [n], \sigma \in \{0,1\}} a_i^\sigma x_i^\sigma \geq A; \rho) < a_{i^*}^{\sigma^*}$ (i.e., add the literals to the current assignment ρ).

ρ	$\text{slack}(C; \rho)$	Comment
\emptyset	8	Sum of coefficients minus degree of falsity.
(\bar{x}_5)	3	Propagates \bar{x}_4 , since coefficient $>$ slack.
(\bar{x}_5, \bar{x}_4)	3	Propagation does not change slack.
$(\bar{x}_5, \bar{x}_4, \bar{x}_3, x_2)$	-2	Conflict, since slack $<$ 0.

Figure 7.15. Slack of $C \doteq x_1 + 2\bar{x}_2 + 3x_3 + 4\bar{x}_4 + 5x_5 \geq 7$ for different trails ρ .

2. If there is no conflict, find some unassigned literal x_i^σ , decide to set it to true (i.e., add x_i^σ to ρ), and go to step 1.
3. Otherwise, fix some conflicting constraint $\sum_{i \in [n], \sigma \in \{0,1\}} a_i^\sigma x_i^\sigma \geq A$, i.e., such that $\text{slack}(\sum_{i \in [n], \sigma \in \{0,1\}} a_i^\sigma x_i^\sigma \geq A; \rho) < 0$, and resolve it with the propagating constraints found in step 1 in reverse chronological order until we derive a constraint that is the analogue of an asserting clause.
4. Add this constraint to the formula, backtrack to the first point in the trail where the constraint is not falsified, and go to step 1.

From this description it is possible to see why the generalized resolution enters the picture in a natural way — the only time we use linear combinations is in step 3, where there are two constraints that have opposing opinions about what value the current variable under consideration should take.

The simple approach outlined above will not quite work. In what remains of this subsection, we first show an example why it fails, and then discuss what can be done to fix the problem. From the examples in Section 7.3 we can see that an important invariant during CDCL conflict analysis is that the assignment that is “left on the trail” always falsifies the currently derived clause. This means that every derived constraint “explains” the conflict by showing what assignments on the trail are inconsistent, and we can continue the conflict analysis until the derived constraint looks “nice,” at which point the solver learns it and backtracks. The standard concept of “niceness” in CDCL is that the constraint should be *asserting*, i.e., that if we remove further literals from the trail in reverse chronological order until the first time when the learned constraint is not falsified, then at this point the constraint propagates some variable that flips an earlier assignment. When we generalized conflict-driven solving to a pseudo-Boolean setting, we would like the conflict analysis to work in the same way.

As a running example, let us consider the pseudo-Boolean formula consisting of the two constraints

$$C_1 \doteq 2x_1 + 2x_2 + 2x_3 + x_4 \geq 4 \quad (7.41a)$$

$$C_2 \doteq 2\bar{x}_1 + 2\bar{x}_2 + 2\bar{x}_3 \geq 3 \quad (7.41b)$$

(which is just an obfuscated way of writing that a majority of the variables $\{x_1, x_2, x_3\}$ have to be true and false at the same time, but it is a simple enough example that we can use it to illustrate our main points). Note that both constraints have slack 3 under the empty assignment, which is larger than all coefficients, so there are no propagations at decision level 0. Suppose that the solver sets x_1 to false to get the trail

$$\rho_1 = (x_1 \stackrel{\text{DEC}}{\leftarrow} 0) . \quad (7.42a)$$

Now $\text{slack}(C_1; \rho_1) = 1$, which is less than the coefficient 2 of x_2 , so C_1 propagates x_2 to true, yielding

$$\rho_2 = (x_1 \stackrel{\text{DEC}}{\leftarrow} 0, x_2 \stackrel{C_1}{\leftarrow} 1) . \quad (7.42b)$$

For the same reason, x_3 is also propagated to true by C_1 (note that, in contrast to CDCL, the same constraint can propagate several times). For this trail

$$\rho_3 = (x_1 \stackrel{\text{DEC}}{\leftarrow} 0, x_2 \stackrel{C_1}{\leftarrow} 1, x_3 \stackrel{C_1}{\leftarrow} 1) \quad (7.42c)$$

the slack of the other constraint C_2 is $\text{slack}(C_2; \rho_3) = -1$, and we have reached a conflict.

Inspired by CDCL conflict analysis, we take the reason constraint $C_1 \doteq \text{reason}(x_3, \rho_2)$ propagating the last literal x_3 under ρ_2 and resolve it over x_3 with the conflicting constraint C_2 , i.e., performing the derivation step

$$\frac{2x_1 + 2x_2 + 2x_3 + x_4 \geq 4 \quad 2\bar{x}_1 + 2\bar{x}_2 + 2\bar{x}_3 \geq 3}{x_4 \geq 1} , \quad (7.43)$$

to obtain the resolvent $x_4 \geq 1 \doteq \text{resolve}(C_1, C_2, x_3)$. But now we have a problem: the slack of the resolvent with respect to what remains on the trail is $\text{slack}(x_4 \geq 1; \rho_2) = 0$. This is no longer negative, so we have lost the invariant from CDCL that the constraint derived on the conflict side should be conflicting!

Taking a step back to analyse what happened, the reason for this failure is that it is in fact possible to satisfy both constraints C_1 and C_2 by extending ρ_2 with the assignment $x_3 = \frac{1}{2}$. Of course, this is not a Boolean assignment. However, taking linear combinations is not a Boolean rule but is sound also over the reals. For this reason, there is no way we can guarantee that the invariant of a conflicting constraint on the conflict side can be maintained if we use only the generalized resolution rule (7.33). Thus, we need to get some Boolean derivation rule into play.

7.6.2.2. Pseudo-Boolean Conflict Analysis Using Saturation

We will now describe how Chai and Kuehlmann [CK05] adapt conflict analysis to a pseudo-Boolean setting using the saturation rule. Saturation in itself cannot help fix our problem, because both constraints resolved in (7.43) are already saturated, as is the resolvent. But if we combine saturation with weakening of the reason constraint, then (perhaps somewhat counter-intuitively) we can get the conflict analysis to work. When resolving a propagating constraint C_{reason} on the reason side with the currently derived constraint C_{confl} on the conflict side (starting with the violated constraint and working our way back along the trail in reverse chronological order, as described above), we will iterate the following procedure:

1. weaken the current reason constraint C_{reason} on some non-falsified literal ℓ' (other than the literal ℓ propagated by C_{reason} , on which we want to apply resolution) to get $\text{weaken}(C_{\text{reason}}, \ell')$;
2. saturate the weakened constraint to obtain $\text{saturate}(\text{weaken}(C_{\text{reason}}, \ell'))$;
3. compute the resolvent $\text{resolve}(C_{\text{confl}}, \text{saturate}(\text{weaken}(C_{\text{reason}}, \ell')), \ell)$ of the weakened reason constraint with the conflicting constraint C_{confl} over the propagated literal ℓ ;

```

while  $slack(resolve(C_{\text{conf}}, C_{\text{reason}}, \ell); \rho) \geq 0$  do
  |  $\ell' \leftarrow$  literal in  $C_{\text{reason}} \setminus \{\ell\}$  not falsified by  $\rho$ ;
  |  $C_{\text{reason}} \leftarrow$  saturate(weaken( $C_{\text{reason}}, \ell'$ ));
end
return  $C_{\text{reason}}$ ;

```

Figure 7.16. Chai-Kuehlmann reason reduction $\text{reduceSat}(C_{\text{conf}}, C_{\text{reason}}, \ell, \rho)$.

until obtaining a resolved constraint that is conflicting. After each step in this iteration C_{reason} is updated to to $\text{weaken}(C_{\text{reason}}, \ell')$, and when we finally get a conflicting resolvent we update C_{conf} to be this new constraint.

Let us first show how this works for our example, and then discuss why this is a correct approach in general. If we weaken $\text{reason}(x_3, \rho_2) \doteq C_1$ on x_2 , which is the first non-falsified literal that is not the one currently propagated, then we get the following derivation:

$$\begin{array}{l}
 \text{Weakening on } x_2 \quad \frac{2x_1 + 2x_2 + 2x_3 + x_4 \geq 4}{2x_1 + 2x_3 + x_4 \geq 2} \\
 \text{Saturation} \quad \frac{2x_1 + 2x_3 + x_4 \geq 2}{2x_1 + 2x_3 + x_4 \geq 2} \qquad 2\bar{x}_1 + 2\bar{x}_2 + 2\bar{x}_3 \geq 3 \\
 \text{Resolution on } x_3 \quad \frac{2x_1 + 2x_3 + x_4 \geq 2}{2\bar{x}_2 + x_4 \geq 1} \qquad 2\bar{x}_1 + 2\bar{x}_2 + 2\bar{x}_3 \geq 3
 \end{array} \tag{7.44}$$

Unfortunately, this does not solve the problem, since $2\bar{x}_2 + x_4 \geq 1$ again has slack 0 with respect to the trail ρ_2 in (7.42b).

We cannot weaken away x_3 , since this is the propagating literal we want to resolve over, but we can weaken C_1 also on x_4 , which is not falsified. This yields

$$\begin{array}{l}
 \text{Weakening on } x_2 \quad \frac{2x_1 + 2x_2 + 2x_3 + x_4 \geq 4}{2x_1 + 2x_3 + x_4 \geq 2} \\
 \text{Weakening on } x_4 \quad \frac{2x_1 + 2x_3 + x_4 \geq 2}{2x_1 + 2x_3 \geq 1} \\
 \text{Saturation} \quad \frac{2x_1 + 2x_3 \geq 1}{x_1 + x_3 \geq 1} \qquad 2\bar{x}_1 + 2\bar{x}_2 + 2\bar{x}_3 \geq 3 \\
 \text{Resolution on } x_3 \quad \frac{x_1 + x_3 \geq 1}{2\bar{x}_2 \geq 1} \qquad 2\bar{x}_1 + 2\bar{x}_2 + 2\bar{x}_3 \geq 3
 \end{array} \tag{7.45}$$

and now we have $slack(2\bar{x}_2 \geq 1; \rho_2) = -1 < 0$, i.e., we have derived a new constraint that maintains the invariant of having negative slack with respect to what remains on the trail. This does not change if we saturate this constraint to get $\bar{x}_2 \geq 1$.

Although we have not formally defined anything like 1UIP pseudo-Boolean constraints — and, indeed, doing so requires some care — it should be clear that $\bar{x}_2 \geq 1$ is asserting. If we undo all decisions on the trail, then at top level we have $slack(\bar{x}_2 \geq 1; \emptyset) = 0$, so x_2 propagates to false yielding $\rho_4 = (x_2 \stackrel{\bar{x}_2 \geq 1}{\leftarrow} 0)$. Looking at $C_1 \doteq 2x_1 + 2x_2 + 2x_3 + x_4 \geq 4$, we have $slack(C_1; \rho_4) = 1$, and since this is smaller than the coefficient 2 of x_1 and x_3 both variables propagate to true. This causes a conflict with $C_2 \doteq 2\bar{x}_1 + 2\bar{x}_2 + 2\bar{x}_3 \geq 3$, and since no decisions have been made the solver can terminate and report that the formula consisting of the constraints (7.41a) and (7.41b) is indeed unsatisfiable.

The key to the pseudo-Boolean conflict analysis just described is that we apply a *reduction algorithm* on the reason constraint, combining weakening and

```

while  $C_{\text{conf}}$  not asserting do
   $\ell \leftarrow$  literal assigned last on trail  $\rho$ ;
  if  $\bar{\ell}$  occurs in  $C_{\text{conf}}$  then
     $C_{\text{reason}} \leftarrow$  reason( $\ell, \rho$ );
     $C_{\text{reason}} \leftarrow$  reduceSat( $C_{\text{reason}}, C_{\text{conf}}, \ell, \rho$ );
     $C_{\text{conf}} \leftarrow$  resolve( $C_{\text{conf}}, C_{\text{reason}}, \ell$ );
     $C_{\text{conf}} \leftarrow$  saturate( $C_{\text{conf}}$ );
  end
   $\rho \leftarrow$  removeLast( $\rho$ );
end
return  $C_{\text{conf}}$ ;

```

Figure 7.17. Pseudo-Boolean conflict analysis $\text{analyzePBconflict}(C_{\text{conf}}, \rho)$.

saturation, to ensure that when the reduced reason constraint is resolved with the currently derived conflict constraint, then the result will be a new conflicting constraint. The pseudo-code for this reduction algorithm from [CK05] is given in Figure 7.16. But how do we know that this procedure is guaranteed to work?

Briefly, the reason is that slack is *subadditive*, i.e., if we take a linear combination of two constraints C and D , then it is not hard to verify that

$$\text{slack}(c \cdot C + d \cdot D; \rho) \leq c \cdot \text{slack}(C; \rho) + d \cdot \text{slack}(D; \rho) \quad (7.46)$$

holds. By the invariant, we know for the currently derived constraint C_{conf} on the conflict side that we have $\text{slack}(C_{\text{conf}}; \rho) < 0$. It is also easy to see directly from the definition (7.38) that weakening the reason constraint C_{reason} leaves $\text{slack}(C_{\text{reason}}; \rho)$ unchanged, since we only weaken on non-falsified literals. But saturation can decrease the slack, and if we have not reached non-positive slack before, then at the very latest this will happen when all non-falsified literals except the propagating one have been weakened away — at this stage the only coefficient contributing to the slack is that of the propagating literal, and since the constraint is saturated this coefficient must be equal to the degree of falsity, so that the whole constraint has slack 0. (This is exactly what happened in our example.) Plugging this into (7.46), we see that a positive linear combination of zero and a negative number will be negative, and the invariant is maintained.

Using this reason reduction method, the whole pseudo-Boolean conflict analysis algorithm will be as in Figure 7.17. The reduction step highlighted in boldface is new compared to CDCL, but everything else is essentially the same (at least at a high level). So how does our conflict analysis compare to CDCL? Let us just point out three important aspects here, which will motivate some of the discussions later.

1. One difference is how much work needs to be performed at each step. When we resolve a new reason with the current conflict clause in CDCL, then we only have to “tag on” the reason clause to the conflict clause, but we do not have to touch the literals already in the conflict clause. Therefore, the total amount of work during CDCL conflict analysis is linear in the sum of the clause sizes. But in pseudo-Boolean analysis we might have to multiply both the reason and the conflict in order to adjust the coefficients so that

the linear combination in (7.33) is cancelling, and this means that we might have to touch all literals in the constraint on the conflict side over and over again. In the worst case, this will incur an extra linear factor in the running time

2. Because of these multiplications, it can also be the case that the coefficients in the constraints grow very large. If this happens, then the integer arithmetic can get hugely expensive. This can become a very serious problem in practice.
3. Perhaps the most serious problem, though, is that for inputs in CNF this procedure described above degenerates to resolution. All that will happen during the conflict analysis is that trivial resolution derivations will produce new clauses, and so the whole algorithm just becomes CDCL but with much more expensive data structures. Hence, if we take the pigeonhole principle formula encoded in CNF and feed it into a pseudo-Boolean solver using this conflict analysis, then although the formula is easy for cutting planes (as shown in [CCT87]) it will be exponentially hard in practice.

The third issue is perhaps the most important one in that it shows how sensitive pseudo-Boolean solvers can be to details of the encoding. We will return to the question of CNF inputs and discuss it in a bit more detail in Section 7.7.

7.6.2.3. Pseudo-Boolean Conflict Analysis Using Division

Motivated by the above discussion, it seems interesting to consider whether the division rule could be a competitive alternative to saturation. It is known that general cutting planes (consisting of the rules (7.27a)–(7.27c) and (7.34)) is *implicationally complete*, meaning that if a pseudo-Boolean formula implies a certain constraint, then there is a way to derive this constraint.³⁸ This is not true for cutting planes with saturation [VEGC⁺18], i.e., when the saturation rule (7.36b) is substituted for the division rule (7.34). For instance, it can be shown that there is no way to derive the cardinality constraint

$$x + y + z \geq 2 \tag{7.47}$$

from the (translation into pseudo-Boolean constraints of the) CNF encoding

$$x + y \geq 1 \tag{7.48a}$$

$$x + z \geq 1 \tag{7.48b}$$

$$y + z \geq 1 \tag{7.48c}$$

using cutting planes with saturation (even with unrestricted linear combinations). In view of this, it is natural to ask whether the use of division could perhaps yield a stronger conflict analysis algorithm. As mentioned above, using division was proposed in the context of general integer linear programming in *CutSat* [JdM13], although it appears that this approach does not work so well in practice. What we will discuss below is a fairly recent variant of pseudo-Boolean conflict analysis

³⁸This is the analogue of Theorem 7.3.1 for resolution saying that of a clause C is implied by a CNF formula F , then there is a resolution derivation of some clause $C' \subseteq C$ from F .

```

c ← coeff(Creason, ℓ);
while slack(resolve(Cconfl, divide(Creason, c), ℓ); ρ) ≥ 0 do
  | ℓ' ← literal in Creason \ {ℓ} such that ℓ̄j ∉ ρ and c † coeff(C, ℓ');
  | Creason ← weaken(Creason, ℓ');
end
return divide(Creason, c);

```

Figure 7.18. Reduction $\text{reduceDiv}(C_{\text{confl}}, C_{\text{reason}}, \ell, \rho)$ using division.

that uses division instead of saturation, and that does seem to perform very well in practice.

In the division-based conflict analysis method, for each step in the conflict analysis we will iterate the following three actions

1. Weaken the reason constraint C_{reason} on some non-falsified literal ℓ' with coefficient not divisible by the coefficient of the propagating literal ℓ to obtain $\text{weaken}(C_{\text{reason}}, \ell')$.
2. Divide the weakened constraint by the coefficient c of the propagating literal, yielding the constraint $\text{divide}(\text{weaken}(C_{\text{reason}}, \ell'), c)$.
3. Compute the resolvent $\text{resolve}(C_{\text{confl}}, \text{divide}(\text{weaken}(C_{\text{reason}}, \ell'), c), \ell)$ of this constraint with the conflicting constraint C_{confl} over the propagated literal ℓ .

These steps are iterated until the resolvent obtained is conflicting. (Again, the constraint C_{reason} is updated to $\text{weaken}(C_{\text{reason}}, \ell')$ after each step.)

Before arguing about correctness, let us do as we did for the saturation-based method and illustrate how this approach works for the constraints C_1 and C_2 from (7.41a)–(7.41b) with the trail $\rho_3 = (x_1 \stackrel{\text{DEC}}{\leftarrow} 0, x_2 \stackrel{C_1}{\leftarrow} 1, x_3 \stackrel{C_1}{\leftarrow} 1)$ in (7.42c), under which C_2 is conflicting. The first attempt to resolve the reason C_1 for x_3 with the conflict constraint C_2 now yields the derivation

$$\begin{array}{l}
 \text{Weakening on } x_4 \quad \frac{2x_1 + 2x_2 + 2x_3 + x_4 \geq 4}{2x_1 + 2x_2 + 2x_3 \geq 3} \\
 \text{Division by 2} \quad \frac{2x_1 + 2x_2 + 2x_3 \geq 3}{x_1 + x_2 + x_3 \geq 2} \\
 \text{Resolution on } x_3 \quad \frac{x_1 + x_2 + x_3 \geq 2 \quad 2\bar{x}_1 + 2\bar{x}_2 + 2\bar{x}_3 \geq 3}{0 \geq 1}
 \end{array} \quad (7.49)$$

so for this particular example the solver immediately derives contradiction and can terminate with a report that the instance is unsatisfiable!

The pseudo-code for the reason reduction algorithm from [EN18] using division is given in Figure 7.18. Let us sketch the argument why this algorithm is guaranteed to return a reduced reason constraint that will maintain our desired invariant, namely that the resolvent of this constraint with the current constraint on the conflict-side has negative slack with respect to the current trail. Just as in the analysis of the reason reduction algorithm reduceSat using saturation in Figure 7.16, it is sufficient to prove that at some point the slack of the constraint on the reason side is guaranteed to become non-positive. This is sufficient to maintain the conflict analysis invariant of negative slack, since the constraint C_{confl} on the conflict side has negative slack by assumption and slack is subadditive (7.46),

```

c ← coeff(C, ℓ);
foreach literal ℓ' in C do
  | if ℓ' ∉ ρ and c ∤ coeff(C, ℓ') then
  |   | C ← weaken(C, ℓ');
  | end
end
return divide(C, c);

```

Figure 7.19. Reason reduction $\text{roundToOne}(C, \ell, \rho)$ used in *RoundingSat*.

meaning that the resolvent of the reason and conflict constraints also has to have negative slack.

Following the notation in Figure 7.18, let c be the coefficient of the literal ℓ propagated by C_{reason} . By the definition of propagation in (7.40) we have

$$0 \leq \text{slack}(C_{\text{reason}}; \rho) < c, \quad (7.50)$$

and since weakening on non-falsified literals does not change the slack these inequalities hold at all times during the execution of `reduceDiv`. Suppose that we have reached the point in the algorithm when all coefficients of non-falsified literals not divisible by c have been weakened away. Consider what contribution the literals in $\text{divide}(C_{\text{reason}}, c)$ give to the slack. Falsified literals in C_{reason} do not contribute at all, and all remaining non-falsified literals have coefficients divisible by c . Therefore, the slack of the reason constraint is divisible by c , i.e., we have

$$\text{slack}(\text{divide}(C_{\text{reason}}, c); \rho) = \frac{\text{slack}(C_{\text{reason}}; \rho)}{c}, \quad (7.51)$$

and it follows from this and (7.50) that

$$0 \leq \text{slack}(\text{divide}(C_{\text{reason}}, c); \rho) < 1, \quad (7.52)$$

i.e., $\text{slack}(\text{divide}(C_{\text{reason}}, c); \rho) = 0$. This proves the correctness of `reduceDiv`.

We remark that the reason reduction method `roundToOne` used in [EN18], and presented in Figure 7.19, does not weaken away literals one by one, but does the maximal amount of weakening right away. This is guaranteed to maintain the invariant by the proof just outlined above. Also, this method is used not only for reason reduction but is applied more aggressively during the conflict analysis. The pseudo-code for the conflict analysis in *RoundingSat* is presented in Figure 7.20.

It is an interesting question how saturation and division compare when used for pseudo-Boolean solving, but this is currently not very well understood. It is clear from our example (7.49) that division can sometimes be more efficient, but one can also construct crafted benchmarks where it seems that saturation should have the potential to work better [GNY19]. Nevertheless, some preliminary conclusions are that for instances where pseudo-Boolean reasoning does not help, so that a competitive approach would have been to translate to CNF and run a CDCL solver, then the conflict speed, and hence the search speed, is orders of magnitude faster in *RoundingSat* than in *Sat4j* [EN18]. For crafted benchmarks where pseudo-Boolean reasoning appears to be crucial, the number of generated

```

while  $C_{\text{conf}}$  contains no or multiple falsified literals on last level do
  if no current solver decisions then
    | output UNSAT and terminate
  end
   $\ell \leftarrow$  literal assigned last on trail  $\rho$ ;
  if  $\bar{\ell}$  occurs in  $C_{\text{conf}}$  then
    |  $C_{\text{conf}} \leftarrow \text{roundToOne}(C_{\text{conf}}, \bar{\ell}, \rho)$ ;
    |  $C_{\text{reason}} \leftarrow \text{roundToOne}(\text{reason}(\ell, \rho), \ell, \rho)$ ;
    |  $C_{\text{conf}} \leftarrow \text{resolve}(C_{\text{conf}}, C_{\text{reason}}, \ell)$ ;
  end
   $\rho \leftarrow \text{removeLast}(\rho)$ ;
end
 $\ell \leftarrow$  literal in  $C_{\text{conf}}$  last falsified by  $\rho$ ;
return  $\text{roundToOne}(C_{\text{conf}}, \ell, \rho)$ ;

```

Figure 7.20. Pseudo-Boolean conflict analysis in *RoundingSat* using `roundToOne`.

conflicts per second in *RoundingSat* goes down significantly, but the performance is still much better than for pseudo-Boolean solvers using saturation [EGCNV18]. One extra bonus is that the frequent use of division helps keep the coefficients small, so that one can use fixed-precision arithmetic (this of course also assumes that there is code for handling overflow, which is an issue we ignore in the pseudo-code presented here). However, the main problem we identified with saturation-based solvers still remains: for CNF inputs, the algorithm degenerates to a CDCL search for resolution proofs.

7.6.3. More Pseudo-Boolean Rules

Before wrapping up our discussion of pseudo-Boolean solving, we wish to mention some other reasoning rules that are relevant to consider in this context. A natural question to ask is whether general linear constraints are needed to harness the full power of pseudo-Boolean solvers, or whether solvers could equally well work with a more limited set of constraints. One particularly interesting class of linear inequalities are cardinality constraints (7.26). Given a general PB constraint such as, for instance,

$$3x_1 + 2x_2 + x_3 + x_4 \geq 4 \quad , \quad (7.53a)$$

one can compute the least number of literals that have to be true, which results in the constraint

$$x_1 + x_2 + x_3 + x_4 \geq 2 \quad . \quad (7.53b)$$

This is used in the solver *Galena* [CK05], which only learns cardinality constraints. The fact that all learned constraints will be of a particular form can also make other aspects of the algorithm easier. Formally, this *cardinality constraint reduction* rule can be written as

$$\frac{\sum_{i \in [n], \sigma \in \{0,1\}} a_i^\sigma x_i^\sigma \geq A}{\sum_{(i,\sigma) : a_i^\sigma > 0} x_i^\sigma \geq T} \quad T = \min\{|I| : I \subseteq [n], \sum_{i \in I, \sigma} a_i^\sigma \geq A\} \quad . \quad (7.54)$$

Another interesting rule is *strengthening*, which we also introduce by giving an example. Consider again the constraints (7.48a)–(7.48c) and suppose that we set

$x \stackrel{\text{DEC}}{\leftarrow} 0$ and propagate. This yields $y \stackrel{x+y \geq 1}{\leftarrow} 1$ and $z \stackrel{x+z \geq 1}{\leftarrow} 1$, meaning that the final constraint $y + z \geq 1$ is “oversatisfied” by a margin of 1. A closer analysis of this situation leads to the conclusion that we can deduce the constraint $x + y + z \geq 2$ in (7.47), since either x is true, in which case the constraint certainly holds, or else $y + z \geq 1$ is oversatisfied as we just saw. Slightly more formally, the *strengthening rule*, which seems to have been imported by [DG02] from operations research, can be described as follows:

- Suppose that assigning $x_j^{\sigma_j} = 0$ implies that the constraint $\sum_{i \neq j, \sigma} a_i x_i^\sigma \geq A$ has to be oversatisfied by an amount of K .
- Then it is sound to deduce the constraint

$$Kx_j^{\sigma_j} + \sum_{i \neq j, \sigma} a_i x_i^\sigma \geq A + K . \quad (7.55)$$

In theory, using strengthening can allow the solver to recover from bad encodings such as CNF (in our example, we recovered the cardinality constraint (7.47) from the CNF encoding (7.48a)–(7.48c)). In practice, however, it seems hard to get this to work in an efficient way.

A final interesting scenario that we want to discuss is the following. If we have the PB constraints

$$2x + 3y + 2z + w \geq 3 \quad (7.56a)$$

$$2\bar{x} + 3y + 2z + w \geq 3 \quad (7.56b)$$

then by eyeballing we can conclude that

$$3y + 2z + w \geq 3 \quad (7.56c)$$

must hold, since x is either true or false and so can contribute to at most one of the constraints (7.56a) and (7.56b). But an application of the generalized resolution rule on these two constraints instead results in the constraint

$$6y + 4z + 2w \geq 4 , \quad (7.56d)$$

reflecting that this rule also takes the possibility $x = \frac{1}{2}$ into consideration. Applying saturation to (7.56d) yields

$$4y + 4z + 2w \geq 4 \quad (7.56e)$$

and division does not help either since it yields the equivalent constraint

$$2y + 2z + w \geq 2 , \quad (7.56f)$$

which is clearly weaker than (7.56c). As observed by [Goc17], it would be quite convenient to have an implementation of what we can call a *fusion resolution* rule

$$\frac{a_j x_j + \sum_{i \neq j, \sigma} b_i^\sigma x_i^\sigma \geq B \quad a_j \bar{x}_j + \sum_{i \neq j, \sigma} b_i^\sigma x_i^\sigma \geq B'}{\sum_{i \neq j, \sigma} b_i^\sigma x_i^\sigma \geq \min\{B, B'\}} . \quad (7.57)$$

The need for such a rule shows up in some tricky benchmarks in [EGCNV18], but we see no obvious way for cutting planes to perform such reasoning in an efficient manner.

7.6.4. Some Challenges for Pseudo-Boolean Solving

We conclude our review of conflict-driven pseudo-Boolean solving with a discussion of some of the challenges that lie ahead. On the theory side, one difficulty is that there seem to be many more degrees of freedom in PB solving compared to conflict-driven clause learning (CDCL). As we have seen above, there are several different ways of generalizing CDCL to a pseudo-Boolean context, and it seems far from obvious what is the best way to do so.

An example of an interesting question concerning the algorithms we have discussed so far is how much the reason should be weakened in the reduction step. Is it better to weaken iteratively, literal by literal, until the first point in time when the resolvent is conflicting? Or is it better to do as in *RoundingSat* and do all the weakening right away? Another curious difference from CDCL is that sometimes the slack on the conflict side can be so negative that it is possible to just skip a resolution step and still maintain that the conflict-side constraint when the last propagated literal is removed from the trail (so that its status changes from falsified to non-falsified and it starts contributing to the slack). In such a scenario, is it better to skip the resolution step, in order to get fewer applications of the resolution rule over all, and get a more “compact explanation” of the conflict, or is it preferable to always resolve? It seems that one can cook up crafted benchmarks supporting both approaches. Different aspects of pseudo-Boolean conflict analysis have been studied in [LMMW20, LMW20], but there is room for much more work in this area.

This leads to the more general question of whether there is a better approach for conflict analysis than generating (the analogue of) trivial resolution derivations. Note that this question also makes sense for CDCL. The main reason in favour of trivial resolution seems to be that it is simple and runs very fast. But perhaps it could sometimes pay off to be slower and do something smarter? Or, in the opposite direction, could it be that one should not try to learn general pseudo-Boolean constraints, as described above, but instead focus on a more limited form such as cardinality constraints, as done in *Galena* [CK05]?

One reason that we do not have any good answers to these questions is that we do not know too much about how the different subsystems of cutting planes described towards the end of Section 7.6.1 relate to each other in terms of theoretical strength. Some progress has been made in recent papers such as [VEGC⁺18, GNY19], but many open problems still remain. A particularly intriguing question is whether cutting planes with division is strictly stronger than cutting planes with saturation when the linear combination rule is limited to generalized resolution. We will discuss this further in Section 7.7.4.

Among the implementation challenges, one of the most important ones is how to achieve efficient propagation for pseudo-Boolean constraints. For CDCL solvers, a simple but crucial observation is that as long as a disjunctive clause contains two non-falsified literals it cannot propagate, and the famous *two-watched-literals scheme* implementing this approach in state-of-the-art solvers is an important part of the explanation how such solvers can run so fast. This is not true in pseudo-Boolean solving — for a constraint like $\sum_{i=1}^n x_i \geq n - 1$ one has to watch all variables in order to detect propagation, since as soon as any single variable

is set to false all others should propagate to true. The recent paper [Dev20] provides what is currently the state of the art in pseudo-Boolean propagations and discusses in more detail the challenges that remain.

A second major challenge is if and how natural and efficient methods can be designed to recover from bad encodings (such as CNF). Although it is hard to make anything like this into a formal claim, it seems that pseudo-Boolean solvers are more sensitive to the exact encoding of a problem, and also to the presence or absence of extra, potentially superfluous constraints, than CDCL solvers. A recent work reporting progress on implementing efficient pseudo-Boolean search for CNF formulas, assuming that these formulas encode cardinality constraints or that such constraints can play an important part in the proofs, is [EN20], but much work still remains.

Another interesting question is how far the solver should backjump once an asserting constraint has been learned. In contrast to CDCL, a learned constraint can be asserting at several levels, and there are even different options in how to define what an asserting constraint should be depending on whether one wants to apply syntactic or (potentially harder to check) semantic criteria. A somewhat related concern is how to assess the “quality” of different constraints, for instance, when the solver has the choice of either performing or skipping a resolution step, and one would like to know which of the two options seems to yield the better constraint.

A puzzling observation, made in, e.g., [EGCNV18], is that sometimes pseudo-Boolean solvers perform extremely poorly on instances which are infeasible even viewed as linear program relaxations without any integrality constraints. Such instances can be trivial for *mixed integer linear programming (MIP)* solvers such as *Gurobi* [Gur], *CPLEX* [CPL], and *SCIP* [SCI], which will detect infeasibility after solving the LP at the root node, while the pseudo-Boolean solvers get completely lost looking for satisfying Boolean assignments in a search space where there are not even any real-valued solutions. On the other hand, it seems that sometimes MIP solvers can fail badly on instances where learning from failed partial assignments to PB constraints appears to be crucial (and where, for this reason, conflict-driven PB solvers can shine). For this reason, a quite tempting proposition would be to borrow techniques from, or merge techniques with, MIP solvers to obtain pseudo-Boolean solvers that could provide the best of both worlds. Some research in this direction has been initiated in [DGN20], but again it seems that most of the work is still ahead.

In order to further develop pseudo-Boolean solving techniques, however, a final challenge that we wish to highlight is that it would be very desirable to develop efficient and concise pseudo-Boolean proof logging techniques. Although the DRAT format employed for CDCL solvers could also be used to log pseudo-Boolean proofs in theory, in practice the required overhead makes this infeasible with current techniques. It seems that the focus on logging only disjunctive clauses mixes very poorly with what is arguably the main advantage of pseudo-Boolean solvers, namely the stronger format of the constraints they derive. There is some recent work on a proof checker *VeriPB* [GMN20b, Ver19] for pseudo-Boolean reasoning, which has been shown to admit efficient proof logging for some constraint programming techniques [EGMN20] and graph solving

algorithms [GMN20a, GMM⁺20], but there is currently no version that would support proof logging for the full range of techniques used by PB solvers in, e.g., the latest pseudo-Boolean solver competition [Pse16].

7.7. Cutting Planes and Proof Complexity

In the previous section we defined different flavours of the cutting planes proof system, and then showed how the conflict-driven SAT solving framework could be lifted from clauses to general 0-1 integer linear constraints. We now wish to proceed to discussing what is known about cutting planes from the point of view of proof complexity.

Before doing so, however, it will be helpful to describe a particular way of constructing crafted benchmarks that has played a crucial role in many of the recent advances on the theoretical side, namely *lifting*. The formal definition of lifted formulas seems to have appeared first in [BHP10], though our discussion below relies heavily on the later paper [HN12]. Readers more interested in seeing statements of concrete proof complexity results can safely skip ahead to Section 7.7.2.

7.7.1. Brief Detour: Lifted CNF Formulas

The idea behind lifting of functions is that we can take a base function f over some domain (which we will choose to be Boolean in this discussion) and extend it to a function over tuples from the same domain by combining it with an *indexing* or *selector* function that determines on which coordinates from the tuples f should be evaluated. More formally, given a positive integer $\ell \geq 2$ and a function $f : \{0, 1\}^m \rightarrow Q$ for some range Q , the *lift of length ℓ of f* is the function $Lift_\ell(f) : \{0, 1\}^{m \times \ell} \times [\ell]^m \rightarrow Q$ such that for any bit-vector $\{x_{i,j}\}_{i \in [m], j \in [\ell]}$ and any $y \in [\ell]^m$ the value of the lifted function is

$$Lift_\ell(f)(x, y) = f(x_{1,y_1}, x_{2,y_2}, \dots, x_{m,y_m}) . \quad (7.58)$$

In words, the y -vector selects which coordinates of the x -vector should be fed to f , as illustrated in Figure 7.21. We refer to the coordinates of the y -vector as *selector variables* and the coordinates of the x -vector as *main variables*, and we write

$$\text{select}_y(x) = (x_{1,y_1}, x_{2,y_2}, \dots, x_{m,y_m}) \quad (7.59)$$

to denote the substring of the x -vector selected by the y -coordinates. (With this notation, we have $\text{select}_y(x) = (x_{1,3}, x_{2,1}, x_{3,2}, x_{4,2})$ in Figure 7.21.)

We next extend this definition from functions to *relations*, or *search problems*. To this end, let S be any search problem defined as a subset of $\{0, 1\}^m \times Q$; that is, on any input $a \in \{0, 1\}^m$, the problem is to find some $q \in Q$ such that $(a, q) \in S$. Then we define the *lift of length ℓ of S* to be a new search problem $Lift_\ell(S) \subseteq \{0, 1\}^{m \times \ell} \times [\ell]^m \times Q$ with input domain $\{0, 1\}^{m \times \ell} \times [\ell]^m$ and output range Q such that for any bit-vector $\{x_{i,j}\}_{i \in [m], j \in [\ell]}$, any $y \in [\ell]^m$, and any $q \in Q$, it holds that

$$(x, y, q) \in Lift_\ell(S) \quad \text{if and only if} \quad (\text{select}_y(x), q) \in S . \quad (7.60)$$

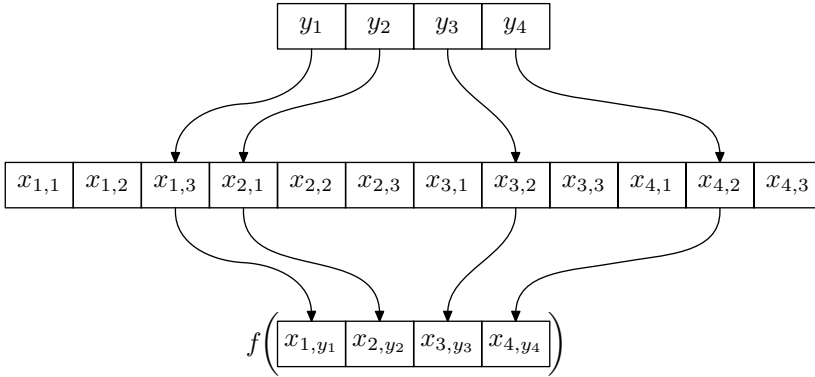


Figure 7.21. Illustration of lifted function (for $\ell = 3$, $m = 4$, and $y = (3, 1, 2, 2)$).

The general paradigm of lifting is to prove that if the function f or the search problem S is hard to compute in some weak computational model, then applying lifting produces a new function $Lift_\ell(f)$ or search problem $Lift_\ell(S)$ that is hard for a much stronger computational models (in which the original problems f or S might be trivial). This is sometimes referred to as *hardness escalation*.

The key behind several recent results establishing lower bounds for the cutting planes proof system has been to study lifted search problems defined in terms of CNF formulas using tools from the area of *communication complexity* [KN97, RY20]. Such lifted search problems are not themselves CNF formulas syntactically speaking, however, and therefore an additional step is needed where these lifted search problems are encoded back into CNF. Such *lifted CNF formulas*, as first introduced in [BHP10], can be constructed in the following way.

Definition 7.7.1 (Lift of CNF formula [BHP10]). Given any CNF formula F with clauses C_1, \dots, C_m over variables u_1, \dots, u_n , and any positive integer $\ell \geq 2$, the *lift of length ℓ* of F is a CNF formula $Lift_\ell(F)$ over $2\ell n$ variables denoted by $\{x_{i,j}\}_{i \in [n], j \in [\ell]}$ (*main variables*) and $\{y_{i,j}\}_{i \in [n], j \in [\ell]}$ (*selector variables*), consisting of the following clauses:

- For every $i \in [n]$, an *auxiliary clause*

$$y_{i,1} \vee y_{i,2} \vee \dots \vee y_{i,\ell} . \quad (7.61a)$$

- For every clause $C_i \in F$, where $C_i = u_{i_1} \vee \dots \vee u_{i_s} \vee \bar{u}_{i_{s+1}} \vee \dots \vee \bar{u}_{i_{s+t}}$ for some $i_1, \dots, i_{s+t} \in [n]$, and for every tuple $(j_1, \dots, j_{s+t}) \in [\ell]^{s+t}$, a *main clause*

$$\bar{y}_{i_1, j_1} \vee x_{i_1, j_1} \vee \dots \vee \bar{y}_{i_s, j_s} \vee x_{i_s, j_s} \vee \bar{y}_{i_{s+1}, j_{s+1}} \vee \bar{x}_{i_{s+1}, j_{s+1}} \vee \dots \vee \bar{y}_{i_{s+t}, j_{s+t}} \vee \bar{x}_{i_{s+t}, j_{s+t}} \quad (7.61b)$$

(where we will refer to C_i as the *original clause* corresponding to the *lifted clause* in (7.61b)).

Let us try to decipher what the notation in Definition 7.7.1 means. The purpose of the auxiliary clauses in (7.61a) is to make sure that in every variable subset $\{y_{i,j} \mid 1 \leq j \leq \ell\}$ at least one of the variables is true. We can think of the selector variables as encoding the vector $y \in [\ell]^m$ in the lifted search problem above. Since every pair $\bar{y}_{i,j} \vee x_{i,j}$ in a main clause (7.61b) is equivalent to an implication $y_{i,j} \rightarrow x_{i,j}$, we can rewrite (7.61b) as

$$(y_{i_1, j_1} \rightarrow x_{i_1, j_1}) \vee \cdots \vee (y_{i_{s+t}, j_{s+t}} \rightarrow \bar{x}_{i_{s+t}, j_{s+t}}) . \quad (7.62)$$

Now we can see that for every clause C_i , the auxiliary clauses encode that there is some choice of selector variables $y_{i,j}$ which are all true, and for this choice of selector variables the $x_{i,j}$ -variables in the lifted clause give us back the original clause C_i . It is easily verified that F is unsatisfiable if and only if $G = \text{Lift}_\ell(F)$ is unsatisfiable, and that if F is a k -CNF formula with m clauses, then G is a $\max(2k, \ell)$ -CNF formula with at most $m\ell^k + n$ clauses. In Figure 7.22 we show the lifted formula $\text{Lift}_2(\text{Peb}_{\Pi_2})$ of length 2 obtained from the pebbling contradiction in Figure 7.11b.³⁹

We remark that there are also other types of lifted CNF formulas in the literature, but Definition 7.7.1 (which, formally speaking, describes lifting using the *indexing gadget*), is the most commonly used version. In the most general sense, though, the pebbling formula with XOR-substitution in Figure 7.12 is also a form of lifted formula.

7.7.2. Cutting Planes Size, Length, and Space

Returning from the detour and focusing again on the cutting planes proof system, the *length* of a cutting planes refutation is the total number of lines/inequalities in it, and the *size* also sums the sizes of all coefficients (i.e., the bit size of representing them). The natural generalization of clause space in resolution is to define (*line*) *space* as the maximal number of linear inequalities needed in memory during a refutation, since every clause is translated into a linear inequality. There is no useful analogue of width/degree known for cutting planes.⁴⁰

Cutting planes can simulate resolution efficiently with respect to length/size and space simultaneously by mimicking the resolution steps one by one. Hence, just as for polynomial calculus we can conclude that the worst-case upper bounds for resolution carry over to cutting planes.

The cutting planes proof system is strictly stronger than resolution with respect to length and size, since cutting planes can refute pigeonhole principle formulas (7.7a)–(7.7b) efficiently [CCT87]. The reason for this is that in contrast to resolution (and polynomial calculus), cutting planes can count. At a high level, pigeonhole principle formulas are refuted simply but summing up the number of

³⁹A very closely related CNF formula, but complemented with clauses $\bar{y}_{i,j} \vee \bar{y}_{i,j'}$ for $1 \leq j < j' \leq \ell$ to enforce that assignments satisfying clauses (7.61a) set only one variable $y_{i,j}$ to true, can be generated in DIMACS format by the tool *CNFgen* [LENV17, CNF] with the command `line cnfgen peb pyramid $\langle h \rangle$ -T lift 2` (for pyramid graphs of height h). Whether these extra clauses are added or not typically does not affect the proof complexity bounds we can obtain for lifted formulas.

⁴⁰That is, one can certainly define width measures, but no such measure is known to have any interesting relation to other complexity measures for cutting planes.

$$\begin{array}{l}
(y_{u,1} \vee y_{u,2}) \\
\wedge (y_{v,1} \vee y_{v,2}) \\
\wedge (y_{w,1} \vee y_{w,2}) \\
\wedge (y_{x,1} \vee y_{x,2}) \\
\wedge (y_{y,1} \vee y_{y,2}) \\
\wedge (y_{z,1} \vee y_{z,2}) \\
\wedge (\bar{y}_{u,1} \vee x_{u,1}) \\
\wedge (\bar{y}_{u,2} \vee x_{u,2}) \\
\wedge (\bar{y}_{v,1} \vee x_{v,1}) \\
\wedge (\bar{y}_{v,2} \vee x_{v,2}) \\
\wedge (\bar{y}_{w,1} \vee x_{w,1}) \\
\wedge (\bar{y}_{w,2} \vee x_{w,2}) \\
\wedge (\bar{y}_{u,1} \vee \bar{x}_{u,1} \vee \bar{y}_{v,1} \vee \bar{x}_{v,1} \vee \bar{y}_{x,1} \vee x_{x,1}) \\
\wedge (\bar{y}_{u,1} \vee \bar{x}_{u,1} \vee \bar{y}_{v,1} \vee \bar{x}_{v,1} \vee \bar{y}_{x,2} \vee x_{x,2}) \\
\wedge (\bar{y}_{u,1} \vee \bar{x}_{u,1} \vee \bar{y}_{v,2} \vee \bar{x}_{v,2} \vee \bar{y}_{x,1} \vee x_{x,1}) \\
\wedge (\bar{y}_{u,1} \vee \bar{x}_{u,1} \vee \bar{y}_{v,2} \vee \bar{x}_{v,2} \vee \bar{y}_{x,2} \vee x_{x,2}) \\
\wedge (\bar{y}_{u,2} \vee \bar{x}_{u,2} \vee \bar{y}_{v,1} \vee \bar{x}_{v,1} \vee \bar{y}_{x,1} \vee x_{x,1}) \\
\wedge (\bar{y}_{u,2} \vee \bar{x}_{u,2} \vee \bar{y}_{v,1} \vee \bar{x}_{v,1} \vee \bar{y}_{x,2} \vee x_{x,2}) \\
\wedge (\bar{y}_{u,2} \vee \bar{x}_{u,2} \vee \bar{y}_{v,2} \vee \bar{x}_{v,2} \vee \bar{y}_{x,1} \vee x_{x,1}) \\
\wedge (\bar{y}_{u,2} \vee \bar{x}_{u,2} \vee \bar{y}_{v,2} \vee \bar{x}_{v,2} \vee \bar{y}_{x,2} \vee x_{x,2}) \\
\wedge (\bar{y}_{v,1} \vee \bar{x}_{v,1} \vee \bar{y}_{w,1} \vee \bar{x}_{w,1} \vee \bar{y}_{y,1} \vee x_{y,1}) \\
\wedge (\bar{y}_{v,1} \vee \bar{x}_{v,1} \vee \bar{y}_{w,1} \vee \bar{x}_{w,1} \vee \bar{y}_{y,2} \vee x_{y,2}) \\
\wedge (\bar{y}_{v,1} \vee \bar{x}_{v,1} \vee \bar{y}_{w,2} \vee \bar{x}_{w,2} \vee \bar{y}_{y,1} \vee x_{y,1}) \\
\wedge (\bar{y}_{v,1} \vee \bar{x}_{v,1} \vee \bar{y}_{w,2} \vee \bar{x}_{w,2} \vee \bar{y}_{y,2} \vee x_{y,2}) \\
\wedge (\bar{y}_{v,2} \vee \bar{x}_{v,2} \vee \bar{y}_{w,1} \vee \bar{x}_{w,1} \vee \bar{y}_{y,1} \vee x_{y,1}) \\
\wedge (\bar{y}_{v,2} \vee \bar{x}_{v,2} \vee \bar{y}_{w,1} \vee \bar{x}_{w,1} \vee \bar{y}_{y,2} \vee x_{y,2}) \\
\wedge (\bar{y}_{v,2} \vee \bar{x}_{v,2} \vee \bar{y}_{w,2} \vee \bar{x}_{w,2} \vee \bar{y}_{y,1} \vee x_{y,1}) \\
\wedge (\bar{y}_{v,2} \vee \bar{x}_{v,2} \vee \bar{y}_{w,2} \vee \bar{x}_{w,2} \vee \bar{y}_{y,2} \vee x_{y,2}) \\
\wedge (\bar{y}_{x,1} \vee \bar{x}_{x,1} \vee \bar{y}_{y,1} \vee \bar{x}_{y,1} \vee \bar{y}_{z,1} \vee x_{z,1}) \\
\wedge (\bar{y}_{x,1} \vee \bar{x}_{x,1} \vee \bar{y}_{y,1} \vee \bar{x}_{y,1} \vee \bar{y}_{z,2} \vee x_{z,2}) \\
\wedge (\bar{y}_{x,1} \vee \bar{x}_{x,1} \vee \bar{y}_{y,2} \vee \bar{x}_{y,2} \vee \bar{y}_{z,1} \vee x_{z,1}) \\
\wedge (\bar{y}_{x,1} \vee \bar{x}_{x,1} \vee \bar{y}_{y,2} \vee \bar{x}_{y,2} \vee \bar{y}_{z,2} \vee x_{z,2}) \\
\wedge (\bar{y}_{x,2} \vee \bar{x}_{x,2} \vee \bar{y}_{y,1} \vee \bar{x}_{y,1} \vee \bar{y}_{z,1} \vee x_{z,1}) \\
\wedge (\bar{y}_{x,2} \vee \bar{x}_{x,2} \vee \bar{y}_{y,1} \vee \bar{x}_{y,1} \vee \bar{y}_{z,2} \vee x_{z,2}) \\
\wedge (\bar{y}_{x,2} \vee \bar{x}_{x,2} \vee \bar{y}_{y,2} \vee \bar{x}_{y,2} \vee \bar{y}_{z,1} \vee x_{z,1}) \\
\wedge (\bar{y}_{x,2} \vee \bar{x}_{x,2} \vee \bar{y}_{y,2} \vee \bar{x}_{y,2} \vee \bar{y}_{z,2} \vee x_{z,2}) \\
\wedge (\bar{y}_{z,1} \vee \bar{x}_{z,1}) \\
\wedge (\bar{y}_{z,2} \vee \bar{x}_{z,2})
\end{array}$$

Figure 7.22. Lifting of the pebbling contradiction in Figure 7.11b.

pigeons and holes, after which the observation can immediately be made that there are too many pigeons to fit into the holes. Cutting planes and polynomial calculus are incomparable with respect to size, i.e., for both proof systems one can find hard formulas that are easy for the other system. PHP formulas are an example of formulas that are hard for polynomial calculus but easy for cutting planes. In the other direction, it was recently shown in [GKRS19] that there are formulas that are easy for polynomial calculus (and even Nullstellensatz) over any field, but are hard for cutting planes. These formulas are obtained by lifting as in Section 7.7.1, but the construction is a bit too involved to describe here.

The length measure in cutting planes does not consider the size of the coefficients. It is natural to ask if, and if so how, the power of cutting planes changes when coefficients are required to be of limited size. In [BC96] it was shown that the size of the coefficients need not be larger than of exponential magnitude if one is willing to tolerate a possible polynomial blow-up in proof length. One can define a subsystem of cutting planes where all inequalities in the proofs have to have coefficients of at most polynomial magnitude measure in the input size (i.e., coefficients should be representable with a logarithmic number of bits), and this subsystem is sometimes denoted CP^* in the literature. Understanding the power of CP^* remains wide open.

Open Problem 7.14. Decide whether cutting planes with coefficients of polynomial magnitude (CP^*) can simulate general cutting planes with at most a polynomial blow-up in proof length, or whether there are formulas for which cutting planes with unbounded coefficients is superpolynomially stronger.

When it comes to space, cutting planes is very much stronger than both resolution and polynomial calculus — it was shown in [GPT15] that any unsatisfiable CNF formula (and, in fact, any set of inconsistent 0-1 linear inequalities) can be refuted in constant line space 5 by cutting planes!⁴¹ This proof works by starting with a linear inequality/hyperplane that cuts away the all-zero point of the Boolean hypercube $\{0, 1\}^n$ from the candidate list of satisfying assignments (there has to exist a clause falsified by this assignment, from which the hyperplane can be obtained), and then uses 4 auxiliary hyperplanes to remove further points $\alpha \in \{0, 1\}^n$ one by one in lexicographical order until all possible assignments have been eliminated, showing that the formulas is unsatisfiable. During the course of this refutation the size of the coefficients of the hyperplanes become exponentially large, however, which the line space measure does not charge for. A very interesting question is what happens if coefficients are limited to be of polynomial magnitude, i.e., if we consider the space complexity of CP^* refutations.

Open Problem 7.15. Determine whether cutting planes with coefficients of polynomial magnitude (CP^*) is as strong as general cutting planes with respect to space, or whether there exist families of formulas that require superconstant space in CP^* .

This question is completely open, and it cannot currently be ruled out that line space 5 would be sufficient. All that is known is that if we restrict the cutting planes proofs to have coefficients of at most constant size, then there are formulas that require $\Omega(\log \log \log n)$ space [GPT15].

If also coefficient sizes are counted, i.e., if one measures the *total space* of cutting planes refutations, then it is not hard to show a linear lower bound (for instance by combining [BSW01] and [BNT13]) and a quadratic worst-case upper bound is immediately implied by resolution. For resolution this quadratic upper bound is known to be tight by [BGT14], but to the best of our knowledge no superlinear lower bounds are known on total space in cutting planes.

Proving space lower bounds, if they exist, seems challenging. It might be worth noting in this context that already cutting planes with coefficients of absolute size 2 (which is the minimum needed to simulate resolution) is quite powerful — this is sufficient to construct space-efficient refutations of pigeonhole principle formulas [GPT15] (when space is measured as the number of inequalities in memory).

For a long time, essentially the only formulas that were known to be hard for the cutting planes proof system with respect to length/size were the *clique-cocklique formulas* (also referred to as *clique-colouring formulas*) claiming (the negation of) that “a graph with an m -clique cannot be $(m - 1)$ -colourable.”

⁴¹Recall that this means that the formula is kept on a read-only input memory, and the working memory never contain more than 5 inequalities at any given time.

The formula consists of clauses

$$q_{k,1} \vee q_{k,2} \vee \cdots \vee q_{k,n} \quad [\text{some vertex is the } k\text{th member of the clique}] \quad (7.63a)$$

$$\bar{q}_{k,i} \vee \bar{q}_{k',i} \quad [\text{clique members are uniquely defined}] \quad (7.63b)$$

$$p_{i,j} \vee \bar{q}_{k,i} \vee \bar{q}_{k',j} \quad [\text{clique members are neighbours}] \quad (7.63c)$$

$$r_{i,1} \vee r_{i,2} \vee \cdots \vee r_{i,m-1} \quad [\text{every vertex } i \text{ has a colour}] \quad (7.63d)$$

$$\bar{p}_{i,j} \vee \bar{r}_{i,\ell} \vee \bar{r}_{j,\ell} \quad [\text{neighbours have distinct colours}] \quad (7.63e)$$

where variables $p_{i,j}$ are indicators of the edges in an n -vertex graph, variables $q_{k,i}$ identify the members of an m -clique in the graph, and variables $r_{i,\ell}$ specify a colouring of the vertices, for indices ranging over $1 \leq i < j \leq n$, $1 \leq k < k' \leq m$, and $1 \leq \ell \leq m - 1$.⁴²

Pudlák [Pud97] proved that these formulas are hard by using a so-called *Craig interpolation* argument, specifically tailored to work for formulas with the right structure. He showed that from any short cutting planes refutation of the formula one can extract a small monotone circuit for clique, which reduces the problem to a question about size lower bounds for monotone circuits. (For completeness, we mention that essentially the same techniques were used in [HC99] to obtain exponential lower bounds for so-called *broken mosquito screen formulas*, but due to space constraints we will not discuss these formulas further here.)

It has been conjectured that Tseitin formulas (as in Figure 7.9) should require long cutting planes refutations, since it should be hard to count mod 2 using linear inequalities. It has seemed even more likely that random k -CNF formulas should be exponentially hard. The last few years have seen some very exciting progress on cutting planes lower bounds. In a breakthrough result, exponential length lower bounds for random CNF formulas of *logarithmic* width were obtained in [HP17, FPPR17]. Unfortunately, currently it does not seem possible to apply the techniques used in these papers to formulas of constant width.

Open Problem 7.16. Prove length lower bounds for cutting planes refutations of random k -CNF formulas for some constant $k \geq 3$.

Another intriguing result established in [GGKS18] is that if one starts with k -CNF formulas (for $k = O(1)$) that require large resolution width, and then applies lifting as described in Section 7.7.1, then this yields formulas which are weakly exponentially hard for cutting planes. Very recently, the cutting planes proof system was shown to be non-automatable, i.e., not to admit efficient proof search, unless $P = NP$ [GKMP20]. As a complement to these results indicating weaknesses of cutting planes, another recent result revealing some of the (quite unexpected) strength of this method of reasoning is that Tseitin formulas have cutting planes refutations of quasipolynomial⁴³ size [DT20]. Interestingly, this

⁴²A closely related version of this formula, but with extra clauses $\bar{q}_{k,i} \vee \bar{q}_{k,j}$ for $1 \leq i < j \leq n$ and $\bar{r}_{i,\ell} \vee \bar{r}_{i,\ell'}$ for $1 \leq \ell < \ell' \leq m - 1$, so that the assignments satisfying the clauses (7.63a) and (7.63d) must set a single variable to true, can be generated by *CNFgen* with the command line `cnfgen cliquecoloring (n) (m) (m-1)` (for the claim that an $(m - 1)$ -colourable n -vertex graph can also have an m -clique). More generally, `cnfgen cliquecoloring (n) (k) (c)` yields a formula claiming that there exists an n -vertex graph that is both c -colourable and has a k -clique. Whether the extra binary clauses discussed above are added or not typically does not affect what proof complexity bounds can be established for these formulas.

⁴³That is, refutations of size $\exp((\log n)^\kappa)$ for some constant κ .

result was obtained by studying the proof system *stabbing planes* [BFI⁺18], which very roughly can be described as an extension of pseudo-Boolean reasoning where one can branch over the truth value not only of variables but of general 0-1 linear inequalities. This proof system is powerful enough to model the kind of reasoning carried out in mixed integer linear programming (MIP) solvers, and appears to be much stronger than cutting planes. However, in some settings it is possible to translate proofs from stabbing planes to cutting planes in a somewhat efficient manner, which is how the result in [DT20] is obtained.

7.7.3. Size-Space Trade-offs for Cutting Planes

Given our very limited understanding of cutting planes, it is perhaps not so surprising that not very much has known about size-space trade-offs for this proof system until quite recently.

It was shown in [GP18a] that short cutting planes refutations of a lifted version of Tseitin formulas on expanders must have large space, but this does not provide a real trade-off since it seems likely that such short refutations do not exist at all, regardless of their space complexity. Earlier, [HN12] proved that short cutting planes refutations of a lifted version of pebbling contradictions (similar to the formula in Figure 7.22) over one particular family of DAGs require large space — a result that was strengthened and generalized by [GP18a] — and for pebbling contradictions such short refutations do exist. Interestingly, and somewhat unexpectedly, all of these results follow from reductions to *communication complexity* [KN97, RY20]. The state of knowledge regarding pebbling contradictions is much worse here than for resolution and polynomial calculus, however — for the latter two proof systems we know of general methods to translate pebbling trade-offs for (essentially) arbitrary graphs into proof complexity size-space trade-offs.

Since [GPT15] established that any unsatisfiable CNF formula has a constant-space refutation, the lower bounds for pebbling contradictions in [HN12, GP18a] yield true size-space trade-off results for cutting planes, with formulas that can be refuted in both small size and small space, but where optimizing both measures simultaneously is impossible. However, the “space-efficient” refutations have coefficients of exponential size. It would be more convincing to obtain trade-offs where the small-space refutations also have small coefficients. Such results would follow if known resolution and polynomial calculus results for pebbling contradictions or Tseitin formulas over long, narrow grids could be extended also to cutting planes, since the resolution refutations establishing the upper bounds in these trade-offs can certainly be simulated by cutting planes with constant-size coefficients. It seems plausible that such trade-off results should hold for these formulas even with respect to cutting planes, but proving this has remained out of reach.

However, using lifted pebbling formulas as in Section 7.7.1, it can be proven that there are formulas which have cutting planes proofs with coefficients of constant size in either small length or small space, but for which any small-space refutation must require superpolynomial length even if the coefficients in the proof are of unbounded size. An example of such a result from [dRNV16] is that there exist CNF formulas $\{F_N\}_{N=1}^{\infty}$ of size $\Theta(N)$ such that:

- F_N can be refuted by cutting planes with constant-size coefficients in size $O(N)$ and (total) space $O(N^{2/5})$.
- F_N can be refuted by cutting planes with constant-size coefficients in (total) space $O(N^{1/40})$ and size $2^{O(N^{1/40})}$.
- Any cutting planes refutation of F_N , even with coefficients of unbounded size, in line space less than $N^{1/20-\epsilon}$ (for some small enough constant $\epsilon > 0$) requires length greater than $2^{\Omega(N^{1/40})}$.

In fact, this trade-off result holds simultaneously for resolution, polynomial calculus, and cutting planes, although for resolution and polynomial calculus the parameters are much worse than in the results surveyed in Sections 7.4.4 and 7.5.6.

As discussed above, determining the precise relation between general cutting planes and the subsystem CP^* with coefficients of at most polynomial magnitude remains open, but what is known regarding Open problems 7.14 and 7.15 can also be stated in the form of a trade-off result. Namely, it was recently shown in [dRMN⁺20] that there are formulas which exhibit strong length-space trade-offs for CP^* but not for general cutting planes. In more detail, it is possible to construct a family of CNF formulas $\{F_n\}_{n=1}^{\infty}$ such that:

- CP^* , and even resolution, can refute F_n in length $\tilde{O}(n)$ (this “big-O-tilde” notation hides polylogarithmic factors in n).
- Cutting planes can refute F_n in length $\tilde{O}(n^2)$ and line space $O(1)$,
- For any CP^* refutation in length L and line space s it must hold that

$$s \cdot \log L = \Omega(n / \log^2 n) . \quad (7.64)$$

In order to make this into a “true trade-off” in the sense of [dRNV16], we would also like to show that CP^* can refute these formulas in small space. This is not known, however, and in fact it is conceivable, or even seems likely, that any CP^* refutation, regardless of the length, would require line space $\Omega(n / \log^2 n)$ for these formulas. Proving such a space lower bound would seem to require new techniques, however.

7.7.4. Subsystems of Cutting Planes

Recall that towards the end of Section 7.6.1 we introduced the following variants of the cutting planes proof system (following [VEGC⁺18]), loosely corresponding to different approaches to pseudo-Boolean solving:

General cutting planes: Rules (7.27a)–(7.27c) and (7.34).

Cutting planes with resolution: As general cutting planes above, except that all applications of (7.27b) and (7.27c) have to be combined into applications of the generalized resolution rule (7.33).

Cutting planes with saturation: Rules (7.27a)–(7.27c) and (7.36b) (with no restrictions on the linear combinations).

Cutting planes with saturation and resolution: As the version of cutting planes with saturation above, except that all applications of the rules (7.27b) and (7.27c) have to be combined instances of generalized resolution (7.33).

As mentioned previously, the general cutting planes proof system is implicationally complete, meaning that if a set of linear inequalities imply some other linear inequality, then the latter inequality can also be derived syntactically. This is *not* true for the other variants of cutting planes listed above. Probably already since [Hoo88, Hoo92] it has been known that cutting planes with division and resolution collapses to the resolution proof system when the input is presented in CNF. It is not hard to show that this holds also if division is replaced by saturation (while restricting linear combinations to general resolution). Thus, subsystems of cutting planes with the generalized resolution rule rather than unrestricted linear combinations can be exponentially weaker than general cutting planes.

However, for cutting planes with saturation it turns out that the generalized resolution rule is not the problem. It was shown in [GNY19], building on [VEGC⁺18], that if the saturation rule is used, then any refutation using unrestricted linear combinations can be transformed into a refutation using only the generalized resolution rule with at most a small polynomial blow-up in the proof length.⁴⁴ Phrased differently, for cutting planes with saturation it is not really a restriction to limit linear combinations to general resolution.

In a pseudo-Boolean solving context it is natural that the generalized resolution rule is used when taking linear combinations of constraints. Such derivation steps arise naturally during conflict analysis, as has been discussed above, and it is hard to see how to devise good heuristics for non-cancelling linear combinations (although it would be very nice if this could be done). A relevant question, therefore, is how the division and saturation rules compare to each other if cutting planes is restricted to generalized resolution. Two results in [GNY19], which we will discuss next, indicate that using division or saturation might lead to incomparable proof systems.

In one direction, there are pseudo-Boolean formulas which have linear-length refutations in cutting planes with division and resolution but require exponential-length refutations in cutting planes with saturation even if there are no restrictions on the linear combinations.

In the other direction, simulating one generalized resolution step followed by a saturation step can take an unbounded number of steps in general cutting planes with division and unrestricted linear combinations. Since the example is simple, we present it here. Let R be a positive integer and consider the two constraints

$$Rx + Ry + \sum_{i=1}^R z_i \geq R \quad (7.65a)$$

and

$$Rx + R\bar{y} + \sum_{i=R+1}^{2R} z_i \geq R . \quad (7.65b)$$

Resolving these two constraints yields

$$2Rx + \sum_{i=1}^{2R} z_i \geq R , \quad (7.66)$$

⁴⁴To be technically precise, note that this is a claim about proof *length* and not about *size*. There is no guarantee in [GNY19] as to what happens to the size of the coefficients.

which after saturation becomes

$$Rx + \sum_{i=1}^{2R} z_i \geq R . \quad (7.67)$$

However, deriving (7.67) from (7.65a) and (7.65b) in general cutting planes (with division and arbitrary linear combinations) can be shown to require $\Omega(\sqrt{R})$ applications of the division rule. Note that this is exponential in the number of bits of the coefficients. It is important to understand that this does *not* show that cutting planes with saturation can be superpolynomially stronger than cutting planes with division — to obtain such a separation, we would need to exhibit a family of pseudo-Boolean formulas for which refutations using division have to be superpolynomially longer than refutations using saturation. What the above example does show, however, is that if CP with division and resolution polynomially simulates CP with saturation and resolution, then such a simulation is unlikely to proceed line by line and instead has to work by some kind of global argument.

Open Problem 7.17. Are there unsatisfiable pseudo-Boolean formulas for which cutting planes with saturation can produce shorter refutations than cutting planes with division?⁴⁵

We can use the above results concerning different versions of cutting planes to understand better the potential and limitations of different approaches to pseudo-Boolean solving.

As we have already discussed, one problem with current pseudo-Boolean solvers is that they perform poorly on CNF inputs. The explanation for this is that if the input is in CNF, then the solvers cannot possibly go beyond resolution, regardless of what heuristics they use. Thus, while solvers that implement native pseudo-Boolean reasoning, such as *Sat4j* [LP10] and *RoundingSat* [EN18], can solve pigeonhole principle formulas very efficiently, they crucially depend on the input being given in pseudo-Boolean form as linear inequalities:

$$p_{i,1} + p_{i,2} + \cdots + p_{i,n} \geq 1 \quad [\text{every pigeon } i \text{ gets a hole}] \quad (7.68a)$$

$$\bar{p}_{1,j} + \bar{p}_{2,j} + \cdots + \bar{p}_{n+1,j} \geq n \quad [\text{no hole } j \text{ gets two pigeons}] \quad (7.68b)$$

If the input is instead presented in CNF, with the cardinality constraints in Equation (7.68b) encoded as the clauses in Equation (7.7b), then the solvers will run in exponential time. The same holds for subset cardinality formulas — if a pseudo-Boolean solver is fed the formula encoded as cardinality constraints, then it runs fast, but on the CNF version in Figure 7.10b it cannot possibly do better than the exponential lower bound on resolution length in [MN14].

An obvious algorithmic challenge is to make pseudo-Boolean solvers reason more efficiently with CNF inputs, so that they could, e.g., detect and use the cardinality constraints hidden in (7.7a)–(7.7b) to get performance comparable to

⁴⁵Here we want the constraints in the pseudo-Boolean formula to be in saturated form, so that the separation would not be a consequence of obfuscated input, but would provide an example where reasoning with saturation during the proof search can be more efficient than reasoning with division, as in the derivation of (7.67) from (7.65a) and (7.65b).

when the input is given as (7.68a)–(7.68b). It is possible to do a preprocessing step to recover cardinality constraints encoded in CNF, and for pigeonhole principle formulas and subset cardinality formulas this works well [BLLM14], but full preprocessing of the input to try to detect cardinality constraints does not seem to be an efficient approach in general. A different idea is to try to perform cardinality constraint detection “on the fly” during the proof search, as proposed in [EN20], but although this seems like a more promising approach it, too, is not yet efficient enough to be used in a general pseudo-Boolean solver. Yet another possibility would be to develop better heuristics for solvers using division and unrestricted linear combinations, since we know that with such rules it is always possible, at least in theory, to rewrite CNF constraints in some other pseudo-Boolean form when this is desirable.

The sensitivity to the input format is not the only challenge in the context of pseudo-Boolean reasoning, however. Another challenging benchmark family are the even colouring (EC) formulas discussed in Section 7.5.4. The pseudo-Boolean encoding of these formulas is obtained by converting every equality constraint to a pair of inequalities, so that, e.g., the first constraint $u + w = 1$ in Figure 7.14b is translated to the inequalities $u + w \geq 1$ and $\bar{u} + \bar{w} \geq 1$, and the second equality $u + z = 1$ is translated to $u + z \geq 1$ and $\bar{u} + \bar{z} \geq 1$, et cetera.

If the number of edges is odd, so that the formula is unsatisfiable, then cutting planes can sum the PB constraints with positive literals over all vertices to derive $2 \cdot \sum_{e \in E(G)} e \geq |E(G)|$ and then divide by 2 and round up to obtain $\sum_{e \in E(G)} e \geq (|E(G)| + 1)/2$. By instead summing up all constraints with negated literals and dividing by 2 one obtains $\sum_{e \in E(G)} \bar{e} \geq (|E(G)| + 1)/2$, and adding these two inequalities cancels all variables and leaves $0 \geq 1$.

One interesting aspect to observe here is that in contrast to pigeonhole principle and subset cardinality formulas, it is essential in the above argument that variables are integer-valued. To see the difference, suppose that we are given a PHP or subset cardinality formula encoded as linear constraints. Then for cutting planes it is sufficient to simply add up the inequalities to derive a contradiction. No integer-based reasoning is needed. Even if we allow putting fractional pigeons into fractional holes, there is no way one can make a pigeon mass of $n + 1$ fit into holes of total capacity n . This set of linear inequalities is unsatisfiable even over the rationals, i.e., the polytope defined by the constraints is empty, and so it is sufficient to just solve the linear programming relaxation. Similarly, for subset cardinality formulas there is no way $4n + 1$ variables could have a total “true mass” of at least $2n + 1$ and a total “false mass” of $2n + 1$ simultaneously. But for collections of linear constraints as in Figure 7.14b, assigning all edges value $\frac{1}{2}$ yields a satisfying fractional solution. The polytope defined by the linear inequalities is *not* empty, but it does not contain any integer points. Hence, refuting EC formulas in cutting planes crucially requires that the solvers use saturation or division (or some other form of integer-based reasoning).

Experiments in [EGCNV18] show that even colouring formulas and some other crafted formulas are much harder for pseudo-Boolean solvers than the cutting planes upper bound would suggest, which seem to indicate that the solvers are still quite far from using the full power of cutting planes reasoning.⁴⁶ It also

⁴⁶To balance this picture somewhat — since this survey chapter might be perceived as taking

seems to be the case that the division-based solver *RoundingSat* performs clearly better than *Sat4j*, and indeed it is not easy to see what a short argument using saturation for the unsatisfiability of even colouring formulas would be. It is tempting to conjecture that EC formulas generated from the right kind of graphs should be exponentially hard for cutting planes with saturation, but proving such a lower bound appears to be beyond current techniques.

Open Problem 7.18. Is it true that even colouring formulas over random 6-regular graphs with an odd number of vertices (so that the formulas are unsatisfiable) are exponentially hard for cutting planes with saturation asymptotically almost surely?

7.7.5. Further Algebraic and Semialgebraic Proof Systems

Nullstellensatz, polynomial calculus, and cutting planes have been generalized to other algebraic and semialgebraic proof systems that are more powerful in several ways. These include the *Lovász-Schrijver* proof system [LS91], the *Sherali-Adams* proof system [SA90], the *Positivstellensatz* proof system [GV01], a system combining Lovász-Schrijver and cutting planes [Pud99], and the *Lasserre* proof system [Las01a]. All of these are methods for reasoning about solutions to polynomial equations. By adapting them to prove the nonexistence of 0-1 solutions for a system of polynomials, they can be used as proof systems for refuting Boolean formulas. There has been extensive work on these systems. For an early expository account of the Positivstellensatz and the Lovász-Schrijver systems, see [GHP02a]. The survey papers [Lau01, CT12] discuss the Lovász-Schrijver, Sherali-Adams, and Lasserre systems.

The *Sum-of-Squares* proof system describes methods for establishing the (non-)existence of general solutions (not just 0-1 solutions) to systems of polynomial equalities over the real numbers. They are similar to the Positivstellensatz system. Sum-of-Squares systems were introduced by [Sho87, Nes00, Par00, Las01b]; see also the survey [BS14]. Other, very strong, algebraic proof systems are the *ideal proof system (IPS)* introduced by Grochow and Pitassi [GP18b] and the noncommutative ideal proof system of [LTW18]. An exposition of these systems is in [PT16].

A recent survey covering proof complexity and Sherali-Adams and Sum-of-Squares is given by [FKP19]. For another survey including the above topics, see [Raz16b]. Not all of the above-discussed systems are propositional proof systems in the sense of Cook and Reckhow; in particular, the validity of IPS proofs can be checked by randomized polynomial time algorithms, but no deterministic polynomial algorithm is known.

An orthogonal family of proof systems — which we mention here anyway because of the connection to cutting planes — are based on *ordered binary decision diagrams (OBDDs)*, which can provide efficient and flexible representations of Boolean functions serving as an alternative to the CNF representation

a rather dim view of the state of the art in pseudo-Boolean solving — we want to stress that there are also applications [LR18, LBD⁺20, SDNS20] where pseudo-Boolean solvers are doing very well, and for some of these applications it seems that the power of the cutting planes proof system is crucial to get better performance than what is offered by CDCL SAT solving, algebraic approaches, constraint programming, and/or mixed integer linear programming.

(see [Bry92]). The paper [AKV04] proposed the use of OBDDs in proof systems, showing they can polynomially simulate resolution and cutting planes with small coefficients. Lower bounds on OBDD-based proof systems were shown in [AKV04], and then by [Kra08, Seg08]. Further lower bounds and simulation results have been obtained in [GZ03, TSZ10, Jĭ1, IKRS17, BIKS18].

7.8. Extended Resolution and DRAT Proof Systems

We now switch our focus to proof systems that are substantially stronger than the earlier considered systems of resolution, Nullstellensatz, polynomial calculus, and cutting planes. Just to give an overview of where we are going, the present section discusses DRAT-style proofs and extended resolution. These systems are directly based on resolution and CDCL solvers, and are intended for use in automated proof systems. Section 7.9 then covers Frege proof systems, which is a generalization of resolution that sits between resolution and extended resolution in proof strength. Extended resolution and Frege proof systems are both stronger than the other proof systems that we have surveyed earlier — in fact, they can give polynomial-size proofs of all the combinatorial principles and counting principles discussed earlier. The only family of formulas discussed so far for which it is not known whether they possess polynomial-size Frege and/or extended resolution proofs are random k -CNF formulas; the usual conjecture is that they do not. Section 7.10 discusses bounded-depth Frege systems, which is a restricted form of Frege systems. Bounded-depth Frege systems are weaker than full Frege systems but stronger than resolution, and are not comparable to cutting planes, in that bounded-depth Frege is more effective on some propositional formulas, but less effective on others.

Our treatment of these topics is theoretical in orientation and relatively brief, partly due to space constraints and partly since these proof systems have yet to be widely used in practical SAT solvers. The exception is that DRAT and similar systems are becoming widely used for proof logging and verification.

Extended resolution (ER) was originally introduced by Tseitin [Tse68] to allow resolution to work with more general formulas than CNFs. The intuition is that an extended resolution refutation should be allowed to introduce proof lines of the form

$$x \leftrightarrow \varphi , \tag{7.69}$$

where x is a *new variable* and φ is an arbitrary propositional formula. The proviso that x is new means that x does not appear in any axiom, in φ , or earlier in the derivation. The extension rule can be used in derivations as well, but then x also must not appear in the formula being derived.

The extension rule (7.69) is stated in the form that will be employed in extended Frege systems, which are defined in Section 7.9. Since resolution systems are constrained to work with clauses, *extended resolution* uses instead a restricted, clausal form of the extension rule. If a and b are literals, and x is a new variable, then the extension rule allows inferring the three clauses

$$\bar{x} \vee a \quad \bar{x} \vee b \quad \bar{a} \vee \bar{b} \vee x \tag{7.70}$$

which express that $x \leftrightarrow (a \wedge b)$. An *extended resolution derivation* is a derivation in which both the resolution rule (7.2) and the extension rule (7.70) are allowed. Using (7.70) multiple times, with multiple new variables, makes it possible to simulate efficiently the action of the full extension inference of (7.69). Consequently, extended resolution simulates (and is simulated by) the extended Frege proof system which uses (7.69).

It is almost immediate that adding the extension rule preserves soundness, since if there is a satisfying assignment to a set of clauses, the satisfying assignment can be extended to give the new variable x the (unique) truth value which satisfies (7.69) or (7.70). Therefore, extended resolution is sound and complete. However, resolution does not polynomially simulate extended resolution. Instead, the addition of extension variables makes the proof system exponentially more powerful. The classic example of this are the pigeonhole principle formulas, which was shown to have polynomial-size extended resolution proofs in [CR79], but to require exponential-size resolution proofs in [Hak85].

The fact that the extension rule is so powerful raises the question of whether practical, CDCL-based, SAT solvers can incorporate the extension rule. If this could be done well, the gains would be enormous, as this would in principle give CDCL the possibility to refute CNFs using the full power of extended resolution. There have been a number of attempts to do so, but the published literature on this is fairly sparse, principally [SB06, AKS10, Hua10, MHB13]. So far, the extension rule has been shown to be useful in limited situations; however, it has not been successful enough to be generally included in SAT solvers. The main bottleneck appears to be that we have no good heuristics for how to choose extension formulas φ for use in the extension rule. A second bottleneck is that even if good choices are made for extension formulas, it appears to be difficult for the CDCL solver to use the new extension variables advantageously for decision literals, unit propagation and clause learning.

In recent years, a new application for the extension rule has appeared, driven by the desire to have SAT solvers output verifiable proofs of correctness of the computations. Section 7.3.1 discussed reverse unit propagation (RUP) proofs as proof traces. RUP proofs, however, are not powerful enough to handle all the preprocessing and inprocessing techniques used by modern SAT solvers. A more powerful *resolution asymmetric tautology* (RAT) inference rule has been developed to supplant RUP, and has later been complemented with a deletion rule in the proof traces to yield the *DRAT* proof system.

The RAT rule differs from RUP rules or resolution in that it does not respect implication. That is to say, it is possible to use a RAT rule to derive a clause C from a set of clauses F even if F does not imply C . Instead, RAT inferences only preserve consistency, in that we are assured $F \cup C$ is consistent if and only if F is. This property is called *equisatisfiability*. As we shall see, examples of this include the pure literal rule and extension axioms. The pure literal rule is used to set a literal that occurs only positively to be set true; the extension axioms are allowed even though they are not true under all truth assignments. Equisatisfiability means that these steps cannot introduce an inconsistency.

Following [HHW13b, HHW13a], a *RAT inference* is formally defined as follows. Let F be the current set of clauses in the clause database. (Recall that

typically the clause database F is repeatedly updated by clause learning and clause deletion.) Let C be a clause $a_1 \vee \cdots \vee a_k \vee b$: here it is permitted, but not required, that the literal b is a new literal not appearing yet in F . Then we may infer C with a *RAT inference* with respect to b , provided that for every clause $D \vee \bar{b}$ in F it holds that the clause

$$a_1 \vee \cdots \vee a_k \vee D \vee b \quad (7.71)$$

either is tautological by virtue of containing clashing literals or is a RUP clause with respect to F . (Recall that a RUP clause is a clause C' which can be inferred by trivial resolution from F , and that this can be checked quickly by adding the negations of the literals in C' to F and using unit propagation to generate a contradiction — see Section 7.3.1 for more details).

The intuition behind a RAT inference is that a clause C of the form $a_1 \vee \cdots \vee a_k \vee b$ can be added to the clause database F without introducing a new inconsistency. One way to justify this is to prove that if the clause C is resolved in all possible ways with clauses of F using the distinguished literal b , then only clauses which are already derivable from F are obtained.

For an example of a RAT inference, let F be the set of clauses

$$\{\bar{b} \vee \bar{a}_1 \vee a_2 \vee a_3, \bar{b} \vee a_4, a_4 \vee \bar{a}_2, a_4 \vee \bar{a}_3, a_2 \vee a_3\} . \quad (7.72)$$

Then the clause $C = a_1 \vee b$ can be inferred by a RAT inference from F with respect to the special variable b . To verify this, note there are two clauses to consider for D by virtue of F containing $D \vee \bar{b}$. The first choice for D is $\bar{a}_1 \vee a_2 \vee a_3$; the resolvent of $D \vee \bar{b}$ with C contains both a_1 and \bar{a}_1 yielding a clause of the form (7.71) that is tautological. The second choice for D is a_4 ; now the resolvent with C is $a_1 \vee a_4$, so the clause (7.71) now becomes $a_1 \vee a_4 \vee b$. This clause is RUP with respect to F since after setting a_1, a_4 and b false in F , we can obtain the empty clause by unit propagation.

Note that the clause C of this example is *not* a consequence of F , since F is satisfied by setting b, a_1, a_4 false and a_2, a_3 true. In particular, RAT can infer C even though $F \not\models C$. Instead, the critical property is that if C can be inferred from F with a RAT inference, then F is satisfiable if and only if $F \cup \{C\}$ is satisfiable. This means that the RAT inference is sound for refutations, but not for derivations. Indeed, the resolution rule is subsumed by the RAT rule, so the RAT rule by itself yields a sound and complete refutation system.

As another example, the RAT inference subsumes the *pure literal rule*. A literal is *pure* if it occurs only positively in the clause database F ; the pure literal rule allows introducing a unit clause asserting the literal is true. Specifically, for the example F of (7.72) the unit clause \bar{a}_1 is trivially RAT with respect to F since there is no clause containing the literal a_1 . Subsequently, the unit clause \bar{b} is RAT as well, for the same reason. In other words, RAT inferences can be used to infer both of the unit clauses \bar{a}_1 and \bar{b} . This is in no way contradicts the previous example, since it is not possible to introduce the clause $b \vee a_1$ at the same time, as the latter clause can no longer be inferred by a RAT inference once \bar{a}_1 and \bar{b} have been introduced. Thus, adding clauses to F may lead to that clauses that were previously RAT clauses are no longer inferable by a RAT inference. Conversely,

deleting clauses from F may cause clauses to become derivable by using the RAT rule.

It is not hard to see that RAT inferences can simulate the extension rule; indeed, the three clauses $\bar{x} \vee a$, $\bar{x} \vee b$, and $\bar{a} \vee \bar{b} \vee x$ of the extension rule (7.70) can be added one at a time by RAT inferences provided F does not contain any clauses containing x or \bar{x} . The first two clauses can be added since F does not contain any clauses involving x ; the third one, because its resolvents with the first two clauses are tautological. Conversely, it is known that extended resolution can simulate the RAT inference [KRPH18]. Therefore, the DRAT proof system and extended resolution polynomially simulate each other.

These simulations between DRAT and extended resolution work because the systems allow introducing arbitrary new variables. Thus, using DRAT for proof search can suffer from the same difficulties as extended resolution; namely, we lack effective methods of deciding what properties the new variables should represent. A different, and very interesting, approach proposed by [HKB17, HKB20] is to consider DRAT without allowing new variables to be introduced. In recent works [HB18, KRPH18, BT19], it has been shown that many of the variants of DRAT are equivalent even when new variables are not allowed to be introduced.

DRAT under the restriction of not introducing new variables is still an area of active research. It is too new to be properly surveyed here, but recent developments include [HHW15, HB18, HKSB17, BT19] and works cited in these papers. An intriguing aspect of DRAT (and related systems) is that, even when no new variables are permitted, it is still unexpectedly strong (see [HKB20, BT19]). It was shown in [BT19, HKB17, HKB20] that even without allowing new variables or clause deletion, a modest extension of the RAT rule called *subset propagation redundancy* (or *SPR* for short), first introduced in [HKB20] is powerful enough to give polynomial-size proofs of many hard examples for bounded-depth Frege proof systems, including pigeonhole principle formulas, clique-colouring formulas, and Tseitin formulas.

The paper [BT19] further showed that when new variables are not allowed, then there is an exponential separation in refutation size between the proof system RAT (the RAT rule without allowing clause deletion) and the proof system DRAT (the RAT rule combined with clause deletion). Clause deletion can give this extra power, since removing clauses permits additional RAT inferences.

An approach to automating the search for DRAT proofs was investigated in the work [HKSB17, HKB19], which used *satisfaction-driven clause learning (SDCL)* to search for short DRAT refutations without new variables. With SDSL, they were able to generate refutations of pigeonhole principle formulas and Tseitin formulas much more quickly than what CDCL can typically accomplish, finding refutations for pigeonhole principle formulas that are close to the optimal polynomial size, although still taking exponential time (at least for PHP formulas). This was accomplished in a very general way without explicitly checking for cardinality constraints. However, so far SDCL has not proved helpful in the general setting of SAT competition problems. As mentioned, there are theoretical results that deletion can exponentially increase the power of (D)RAT; however, there are at present no practical techniques for incorporating this extra power of deletion. In addition, if the deletion rule is not constrained to preserve equisatisfiability,

then deletion may change an unsatisfiable formula into a satisfiable formula. This would be incompatible with present-day CDCL solvers, which search simultaneously for satisfying assignments and proofs of unsatisfiability. For more on DRAT and related systems, see Chapter 15.

7.9. Frege and Extended Frege Proof Systems

Frege proofs are the standard “textbook” propositional proof systems, typically with modus ponens as the sole inference rule, and frequently formulated with connectives \neg , \wedge , \vee , and \rightarrow . In contrast to previous sections, formulas are no longer required to be in CNF, and instead can be formed using arbitrary combinations of Boolean connectives.

Frege systems were first defined in the setting of proof complexity by [CR79, Rec75]). They are axiomatized with a finite set of axiom schemes (for example $F \wedge G \rightarrow F$ is a possible axiom) and a finite set of inference rules (for instance, modus ponens). This allows for many possible Frege systems. Indeed, different sets of connectives, and different axioms and rules of inference can be used. It is required that the connectives of a Frege system can form formulas expressing any Boolean functions (that is that the connectives are “complete”); it is also required that a Frege proof system be implicationaly sound and complete. To explain this last bit, let F be a formula and Γ be a set of formulas; recall that we write $\Gamma \models F$ to indicate that Γ logically implies F . For a Frege system \mathcal{F} to be implicationaly sound and complete, we must have $\Gamma \models F$ if and only if there is an \mathcal{F} -derivation of F from the formulas in Γ . As shown in [CR79, Rec75]), any two Frege systems polynomially simulate each other, so the exact choice of axioms and rules of inference is not particularly important.

We present here an alternative definition of Frege proof systems based on the sequent calculus. This is an elegant and flexible framework for formulating many different proof systems; we describe one particular version, called here *LK*, suitable for propositional logic.⁴⁷ *Propositional formulas* are formed using the propositional connectives \wedge and \vee and variables x and negated variables \bar{x} . Propositional formulas are defined formally by induction: firstly, for x any variable, x and \bar{x} are both propositional formulas and secondly, if φ and ψ are propositional formulas, then their conjunction ($\varphi \wedge \psi$) and their disjunction ($\varphi \vee \psi$) are propositional formulas. Note that propositional formulas allow arbitrary use of conjunctions and disjunctions and consequently are much more general than the CNF formulas considered earlier in this survey.

The lines in an LK proof are sequents. A *sequent* is an expression of the form

$$\varphi_1, \varphi_2, \dots, \varphi_k \Rightarrow \psi_1, \psi_2, \dots, \psi_\ell, \tag{7.73}$$

where the φ_i 's and ψ_i 's are propositional formulas. The intended meaning of this sequent is that the conjunction of the formulas φ_i on the left implies the disjunction of the formulas ψ_j on the right. That is, commas on the left-hand

⁴⁷The name LK is often used instead for a first-order logic, as originally introduced by Gentzen [Gen35]; our propositional system is sometimes called PK in the literature.

side of (7.73) should be understood as if they were \wedge 's, and commas on the right-hand side should be understood as if they were \vee 's. Equivalently, the sequent expresses that the disjunction of the ψ_j 's and the $\neg\varphi_i$'s is true.

It is convenient to use the convention that in the sequent (7.73), the right- and left-hand sides are *multisets*. That is, the order of the formulas $\varphi_1, \dots, \varphi_k$ is unimportant, but the multiplicities of formulas are tracked. We use Γ, Δ, \dots to denote multisets of formulas, and often use commas to denote the union of multisets. The rules of inference for LK are:

Initial sequents: For any variable x , there are three initial sequents (*logical axioms*):

$$x \Rightarrow x \quad \text{and} \quad x, \bar{x} \Rightarrow \quad \text{and} \quad \Rightarrow x, \bar{x} \quad (7.74a)$$

Structural rules:

$$\text{Weak-left: } \frac{\Gamma \Rightarrow \Delta}{\varphi, \Gamma \Rightarrow \Delta} \quad \text{Weak-right: } \frac{\Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta, \varphi} \quad (7.74b)$$

$$\text{Contract-left: } \frac{\varphi, \varphi, \Gamma \Rightarrow \Delta}{\varphi, \Gamma \Rightarrow \Delta} \quad \text{Contract-right: } \frac{\Gamma \Rightarrow \Delta, \varphi, \varphi}{\Gamma \Rightarrow \Delta, \varphi} \quad (7.74c)$$

Cut rule:

$$\text{Cut: } \frac{\Gamma \Rightarrow \Delta, \varphi \quad \varphi, \Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta} \quad (7.74d)$$

Logical rules:

$$\wedge\text{-left: } \frac{\varphi, \psi, \Gamma \Rightarrow \Delta}{\varphi \wedge \psi, \Gamma \Rightarrow \Delta} \quad \wedge\text{-right: } \frac{\Gamma \Rightarrow \Delta, \varphi \quad \Gamma \Rightarrow \Delta, \psi}{\Gamma \Rightarrow \Delta, \varphi \wedge \psi} \quad (7.74e)$$

$$\vee\text{-left: } \frac{\varphi, \Gamma \Rightarrow \Delta \quad \psi, \Gamma \Rightarrow \Delta}{\varphi \vee \psi, \Gamma \Rightarrow \Delta} \quad \vee\text{-right: } \frac{\Gamma \Rightarrow \Delta, \varphi, \psi}{\Gamma \Rightarrow \Delta, \varphi \vee \psi} \quad (7.74f)$$

For extended Frege proofs (but not for Frege or LK), *extension axioms* are also allowed as initial sequents. Extension axioms are specified by a sequence of pairs (x_i, ψ_i) for $i = 1, \dots, k$ so that each ψ_i is a formula and each *extension variable* x_i is a “new” variable that does not appear in the conclusion of the proof and does not appear in any ψ_j for $j \leq i$. Then an extended Frege proof may use the following extension axioms as initial sequents:

Extension axioms:

$$x_i \Rightarrow \psi_i \quad \text{and} \quad \psi_i \Rightarrow x_i \quad (7.75)$$

An LK proof is a sequence of sequents, where each sequent either is a logical axiom or is inferred from earlier sequents by a valid LK rule of inference.⁴⁸ A *Frege proof* of a formula φ is defined to be an LK proof of the sequent $\Rightarrow \varphi$. An *extended Frege*

⁴⁸It is typical in the literature to require LK proofs to be tree-like; we instead allow DAG-like proofs, so as to be consistent with the other systems discussed in this survey. Perhaps confusingly however, Frege proofs based on modus ponens are typically defined to be DAG-like in the literature. This makes no difference for us, as tree-like LK proofs can polynomially simulate (DAG-like) LK proofs [Kra94], so the distinction between tree-like and DAG-like proofs is not important for Frege systems.

proof of φ is defined similarly, but allowing the use of extension axioms (7.75) as initial sequents. Extended Frege is equivalent to extended resolution, in that the two systems polynomially simulate each other.

We can also use LK as a refutation system. For this, let \mathcal{S} be a set of sequents. An LK refutation may use both the logical axioms of (7.74a) and sequents from \mathcal{S} as initial sequents and ends with the empty sequent \Rightarrow . The initial sequents from \mathcal{S} are called *nonlogical axioms*. Since the empty sequent is false under any truth assignment, an LK refutation proves the unsatisfiability of the nonlogical axioms \mathcal{S} .

As already mentioned, Frege systems are strictly stronger than resolution; in fact, even the weaker bounded-depth Frege systems discussed in the next section are more powerful: they can polynomially simulate resolution, but cannot be polynomially simulated by resolution. It was shown in [Bus87] that Frege systems have polynomial-size proofs of the pigeonhole principle formulas, whereas, as already mentioned several times above, the smallest resolution refutations grow exponentially [Hak85].

The polynomial-size Frege proofs of pigeonhole principle formulas can be constructed by showing that Frege systems can define counting, and even define the sum of a vector of integers when the integers are presented as propositional variables representing the integers in binary notation ([Bus87]. This was exploited in [Goe90] to prove that Frege systems can polynomially simulate cutting planes proofs. Furthermore, it is not hard to see that Frege systems can polynomially simulate Nullstellensatz and polynomial calculus over any finite field \mathbb{F} .

Extended Frege systems are conjectured to be yet stronger than Frege systems, but we lack good examples of combinatorial properties that might have short extended Frege proofs but require exponentially long Frege proofs. The paper [BBP95] identified some candidates, but these have largely been ruled out in recent years [HT15, TC17, ABB16, Bus15].

Open Problem 7.19. Find new combinatorial candidates for exponentially separating Frege and extended Frege proofs.

Open Problem 7.20. Give a conditional exponential lower bound for Frege proofs using some assumption weaker than $\text{NP} \neq \text{coNP}$.

7.10. Bounded-Depth Frege Proof System

One important measure of the complexity of general propositional formulas with the connectives \wedge and \vee is the number of alternations of \wedge 's and \vee 's. The *depth* of a propositional formula φ equals the number of levels of disjunctions and conjunctions in φ . More formally, any literal x or \bar{x} is a depth-0 formula, and if $\varphi_1, \dots, \varphi_k$ are all depth strictly smaller than d , then any formula formed by combining those k formulas with only conjunctions, or with only disjunctions, has depth bounded by d . For example, $v \wedge ((x \vee (\bar{y} \vee z)) \wedge \bar{u})$ is a depth-2 formula.

The sequent calculus LK also lends itself to working with Frege proofs with restrictions on formula depth in a natural way. At the bottom level, resolution can be viewed as a depth-0 LK refutation system. In particular, a resolution refutation can be viewed as an LK refutation in which all formulas are merely propositional

variables. To see this, note that if C is a clause containing the variables x_1, \dots, x_k unnegated and the negated literals $\bar{y}_1, \dots, \bar{y}_{k'}$, then the corresponding sequent S_C is $y_1, \dots, y_{k'} \Rightarrow x_1, \dots, x_k$. Under this translation between clauses and sequents of literals, the resolution inference rule corresponds exactly to a cut inference in depth-0 LK. In this way, resolution refutations are essentially identical to depth-0 LK refutations.

The notion of depth-0 LK proof can be generalized to any bounded depth d . For $d \geq 0$, a *depth- d LK proof* is an LK proof in which every formula has depth at most d . We write *d -LK* as a compact notation for the LK proof system restricted to depth- d proofs.

Loosely speaking, the *depth- d Frege system* is the same as the d -LK proof system. However, the notion of depth for Frege proofs is sensitive to issues such as whether proofs are tree-like or DAG-like, and whether proofs are formulated Hilbert-style with modus ponens (say) or with the sequent calculus. Thus, to be formal, we will restrict our attention to the (DAG-like) d -LK proof system.

The *Res(k)* or *k -DNF resolution* proof system is an extension of resolution in which the lines are DNF formulas where the conjunctions have size bounded by k . Note that *Res(1)* is the same as resolution, so *Res(k)* proof systems lie just above resolution in strength. *Res(k)* can also be defined as the subsystem of depth-1 LK in which all formulas are either disjunctions or conjunctions of at most k literals. A number of exponential lower bounds on *Res(k)* refutations of weak pigeonhole principle formulas have been proven in a sequence of papers [ABE02, SBI04, Ale11, AMO15, Raz15] with Razborov [Raz15] establishing exponential lower bounds for $k = \epsilon \log n / \log \log n$. Size-space trade-offs for *Res(k)* were obtained in [BSN11].

There are also exponential lower bounds on the size of depth- d LK proofs for all constant $d > 0$. Haken's exponential lower bound for resolution refutations of pigeonhole principle formulas [Hak85] applies to depth-0 LK. This was substantially extended to give strong lower bounds on the size of bounded-depth LK proofs of the pigeonhole principle formulas by [PBI93, KPW95] who showed that with $m = n + 1$ pigeons and n holes, depth d LK refutations of the PHP formulas require size $\exp(\Omega(n^{5^{1/d}}))$, improving on earlier lower bounds of [Ajt88].

Other known hard principles for bounded-depth LK include the *counting-mod- p principle* formulas.⁴⁹ We define here only the case $p = 2$; this is the *parity principle*. Fix $n > 0$ an odd integer. The parity principle *PARITY $_n$* uses variables $x_{i,j}$ for $1 \leq i < j \leq n$; for convenience of notation, we define $x_{j,i}$ to be the same variable as $x_{i,j}$. The clauses of *PARITY $_n$* consist of the n many totality clauses

$$\bigvee_{j \neq i} x_{i,j} \quad \text{for } 1 \leq i \leq n \quad (7.76a)$$

and the $n \binom{n-1}{2}$ many clauses

$$\bar{x}_{i,j} \vee \bar{x}_{i',j} \quad \text{for } i, i', j \in [n], i \neq i' \neq j \neq i \quad (7.76b)$$

⁴⁹Counting-mod- p principle formulas can be generated in the standard DIMACS format for CNF formulas used by SAT solvers by the tool *CNFgen* [LENV17, CNF] using the command line `cnfgen count <n> <p>` for positive integers n and p such that p does not divide n (in order to get unsatisfiable instances).

claiming that there is a partition of $[n]$ into pairs of elements. These parity principle formulas $PARITY_n$ ⁵⁰ require exponential-size bounded-depth LK refutations; it was established in [BP96a] that depth- d LK refutations of $PARITY_n$ require size $\exp(\Omega(n^{6^{1/(d+1)}}))$. This lower bound holds even if formulas expressing the unsatisfiability of PHP_n^{n+1} are permitted as additional axioms.

The lower bounds for the parity principle refutations can also be generalized to exponential lower bounds for general *counting-mod- p* principles for $p > 2$, as proved by [BIK⁺97] building on work of [BIK⁺96] and [Rii97a, Rii97b].

There are a number of other exponential lower bounds known for bounded-depth LK systems, but these two results are representative of the best known lower bounds. A recent result of [PRST16] has opened up a potential avenue for new lower bounds, by showing a lower bound of $\exp(\Omega((\log n)^2/d^2))$ for the size of d -LK refutations of Tseitin formulas on 3-regular expander graphs where n is the number of vertices in the graph. This result implies a *depth lower bound* of $d = \Omega(\sqrt{\log n})$ for polynomial-size d -LK refutations of these Tseitin formulas. Håstad [Hås17] instead focused on Tseitin formulas defined over grid graphs to obtain an improved size lower bound for d -LK of $\exp(\Omega(n^{1/58(d+1)}))$ and an improved depth lower bound of $\Omega(\log n / \log \log n)$.

The above-discussed results are some of the highlights regarding lower bounds on the size of bounded-depth LK proofs. Along with the lower bounds for polynomial calculus in Section 7.5.2 and cutting planes in Section 7.7.2, these are the state of the art in proving size lower bounds on proof size.

There are also a large number of open problems remaining about the size of bounded-depth LK proofs. Many of these problems lie right at the border of where we expect to be able to prove lower bounds on the complexity of proofs without actually resolving fundamental open questions such as whether $P \neq NP$ or not. One such open problem is to give better separations of depth- d and depth- $(d+1)$ systems.

Open Problem 7.21. Is there an exponential separation between the size of depth- d LK and depth- $(d+1)$ LK refutations of CNF formulas?

So far, only a superpolynomial separation is known for d -LK versus $(d+1)$ -LK proofs when refuting CNF formulas. This separation is obtained by, on the one hand, using the above-discussed lower bounds for d -LK proofs for pigeonhole principle formulas with $n+1$ pigeons and n holes and, on the other hand, constructing small $(d+1)$ -LK proofs of the pigeonhole principle formulas using a speedup technique of Nepomnjaščii [Nep70]. Discussing this in more detail is beyond the scope of this survey; the only proof in the literature is described in [KI02] using formalizations in bounded arithmetic.

One of the reasons that Open Problem 7.21 is so tantalizing is that the Yao-Håstad switching lemmas [Hås86] in circuit complexity tell us that depth- $(d+1)$ propositional formulas can be exponentially more succinct in expressibility than depth- d propositional formulas. This seems like it should give $(d+1)$ -LK proofs more power than d -LK proofs; however, Open Problem 7.21 remains unsolved even though considerable effort has been spent on it.

⁵⁰Parity principle formulas can be generated by *CNFgen* with the command line `cnfgen parity <n>` with the integer n chosen odd (to get unsatisfiable instances).

Another open problem concerns the power of bounded-depth LK when the language is extended to include the parity connective \oplus (exclusive or) in addition to \wedge and \vee . It is straightforward to extend LK to accommodate formulas that use the connectives \wedge , \vee and \oplus ; and the notion of depth extends naturally to the notion of $\{\wedge, \vee, \oplus\}$ -depth. We write $d\text{-LK}(\oplus)$ to denote the depth- d LK system so extended. There are polynomial size, bounded-depth $\text{LK}(\oplus)$ proofs of many principles that depend on only counting mod 2; these include the Tseitin formulas and the onto functional pigeonhole principle formulas for $m = n + 1$ pigeons and n holes. Fundamental results of Razborov [Raz87] and Smolensky [Smo87] in circuit complexity give exponential lower bounds on the expressibility of propositional formulas using the connectives \wedge , \vee and \oplus . For instance the mod-3 summation function cannot be expressed by polynomial-size, bounded-depth propositional formulas with those connectives. Thus, bounded-depth $\text{LK}(\oplus)$ proofs cannot reason directly about mod-3 counting. Nonetheless, it is an open problem to obtain superpolynomial lower bounds for this proof system.

Open Problem 7.22. Let $\text{LK}(\oplus)$ denote LK extended with a parity (exclusive or) logical connective. Are there exponential, or even superpolynomial, lower bounds on the size of bounded-depth $\text{LK}(\oplus)$ proofs? Can bounded-depth $\text{LK}(\oplus)$ be simulated by bounded-depth LK augmented with all instances of formulas expressing the unsatisfiability of the PARITY_n formulas?

Open Problem 7.22 is related to the Nullstellensatz proof system discussed in Section 7.5.1. In fact, the study of proof systems based on counting principles and counting gates was the original impetus for the development of Nullstellensatz [BIK⁺96].

It is interesting to compare the strength of bounded-depth LK proofs to the strengths of the proof systems of resolution, cutting planes, and polynomial calculus. First, as already mentioned, $d\text{-LK}$ polynomially simulates resolution for all $d \geq 0$. However, resolution does not polynomially simulate all the systems $d\text{-LK}$. An example of this are the weak pigeonhole principle formulas PHP_n^{2n} expressing that there is no injection from $2n$ pigeons to n holes. These formulas are known from [PWW88, MPW02] to have quasipolynomial-size 2-LK proofs; but from [Hak85, BT88] to require exponential-size resolution refutations.

Bounded-depth LK systems are not directly comparable to cutting planes. In one direction, there are short cutting planes refutations of PHP and parity principle formulas, for instance. However, these require exponential-size $d\text{-LK}$ proofs for any fixed $d > 0$. For the other direction, an exponential separation can be obtained using a “weak” clique-coclique principle. Recall that the clique-coclique formulas ((7.63a)–(7.63e)) encode that a graph cannot have both a clique of size $m + 1$ and a colouring of size m . By the “weak” clique-coclique formulas we mean an analogous version that states that a graph cannot have clique of size $2m$ and a colouring of size m (this is a weaker claim, and so should be easier to prove). These latter formulas have quasipolynomial-size 3-LK proofs by adapting the quasipolynomial-size 2-LK proofs for weak PHP formulas. On the other hand, the Craig interpolation method of [Pud97], along with lower bounds on monotone real circuits based on [AB87], still work for the weak clique-coclique formulas to show that they require exponential-size cutting planes refutations.

Finally, we can compare bounded-depth Frege to polynomial calculus. In one direction, there are CNF formulas that are easy to refute in polynomial calculus, or even Nullstellensatz, but require exponential-size d -LK proofs for any fixed d , for instance, mod- p counting principles when working over a field of characteristic p . A separation in the other direction follows from [GL10b]. Thus, bounded-depth Frege and polynomial calculus are incomparable proof systems.

7.11. Concluding Remarks

In this chapter we have presented an overview of proof complexity with a focus on connections with SAT solving. On the proof complexity side, one main take-away message is that resolution is fairly well-understood, although substantial important open questions still remain open. There has also been good progress in research on polynomial calculus, cutting planes and bounded-depth Frege systems, but these proof systems are not nearly as well understood as resolution.

When it comes to applied SAT solving (where, as mentioned in the introductory section, we are mostly restricted to studying unsatisfiable formulas, since this is the setting in which the tools from proof complexity can most naturally be brought to bear), there is still lots of room for improvement in our understanding of why different formulas are easy or hard, why different proof systems are better or worse for proof search, and why sometimes seemingly very small differences in algorithms can make big differences in the effectiveness of SAT solvers. It would be interesting to investigate further whether there are relevant connections between proof complexity measures and hardness of SAT, and whether proof complexity can help to shed light on the inner workings on SAT solvers.

We want to stress again that there are many topics in proof complexity that we have not been able to cover in this survey chapter due to the finiteness of time and space, and to a large extent our choice of topics has been guided by considerations as to what should be most relevant in an applied SAT solving context. One major omission from a theoretical point of view is the topic of *bounded arithmetic*. Bounded arithmetic theories are a collection of first-order and second-order theories in formal logic with close connections to both proof complexity and computational complexity theory more broadly. An overview of bounded arithmetic can be found in the books [Bus86, Kra95, CN10] and the final part of [HP93], and the connections to proof complexity are covered in [Kra95, CN10]. The *Paris-Wilkie translation* [PW85] gives a direct connection between bounded arithmetic and quasipolynomial-size, bounded-depth Frege proofs. *Cook translations* [Coo75, Ara00, CM05, CN10, BPT14] relate bounded arithmetic theories to the Frege and extended Frege proof systems.

Another important topic that we have not discussed is *symmetry reasoning* (see Chapter 13). Many of the combinatorial formulas that we have examined in this survey chapter are hard because they contain symmetries that make them hard to deal with for different proof systems — examples include pigeonhole principle formulas for resolution and polynomial calculus, clique-coclique formulas for cutting planes, and (perhaps less obviously) Tseitin formulas for bounded-depth Frege — but these formulas become trivial once reasoning with symmetries is allowed. In practice, different methods of handling symmetries that use static

or dynamic techniques, or combinations thereof, have been developed in, e.g., [DBBD16, DBB17, MBCK18, MBK19, TD19] and have been shown to be quite efficient in many cases, and the question of how to express symmetry reasoning as DRAT proofs has been studied in [HHW15]. It would be very interesting to perform a theoretical study of the strength of the methods of reasoning in the above-mentioned papers. However, we are not aware of any work in proof complexity that focuses on proof systems enhanced with rules for symmetry reasoning apart from a few (by now rather old) papers such as [Kri85, Urq99, AU00, Sze05].

We also want to mention briefly a (somewhat unexpected) connection between SAT solving and MaxSAT solving (discussed in Chapters 23 and 24). In general, it is more difficult to solve MaxSAT problems than SAT problems. However, *dual-rail MaxSAT* has recently been introduced in the papers [IMMS17, BBI⁺18, MIB⁺19] as a method for solving SAT instances. A dual-rail proof system replaces each propositional variable x with two new propositional variables p_x and n_x , for the positive and negative values of x , representing the literals x and \bar{x} . An appropriate dual-rail formulation allows a SAT instance to be reduced to a MaxSAT instance which can be easier in some cases.

Concluding this chapter, the main algorithmic challenge we want to highlight is if and how one can build efficient SAT solvers based on stronger proof systems than resolution. Is it really the case that conflict-driven clause learning (CDCL), originating in the DPLL method from the early 1960s, is the best conceivable paradigm? Or could it be possible that is it now time, over 50 years later, to take the next step and build fundamentally different SAT solvers, perhaps based on some of the algebraic and/or geometric methods discussed in this survey chapter? Is the comparative lack of progress on SAT solvers using such stronger method of reasoning an indication of that there are fundamental limitations to efficient proof search being implemented within stronger proof systems? Or could it be that a sustained long-term effort will yield powerful new SAT solving paradigms, just as the immense work spent on optimizing CDCL solvers over the years have led to improvements in performance of several orders of magnitude?

Acknowledgements

We are most grateful to all our colleagues in the proof complexity and SAT communities, too numerous to list here, with whom we have had stimulating and enlightening discussions over the years. We want to give a special thanks to Paul Beame, Jan Elffers, Stephan Gocht, Massimo Lauria, Gunnar Stålmarck, and Marc Vinyals for answering questions and/or proof-reading different versions of the manuscript, helping to catch many typos and other mistakes, and giving suggestions how to improve the exposition. We are also thankful to Massimo Lauria and Marc Vinyals for providing information on how to generate different crafted proof complexity benchmark formulas using the tool *CNFgen* and finding suitable values for the parameters. Furthermore, we wish to thank Daniel Le Berre, João Marques-Silva, and Dmitry Sokolov for help with literature references, and Armin Biere, Laurent Simon, and Niklas Sörensson for providing empirical data on SAT solver performance. Last, but certainly not least, we are most indebted to Robert Robere for an amazingly thorough review of the first completed version

of the chapter, which further helped to enhance the manuscript. Any remaining errors or deficiencies are certainly the responsibility of the authors only.

Part of this work was carried out while the second author visited the Simons Institute for the Theory of Computing at UC Berkeley in association with the DIMACS/Simons Collaboration on Lower Bounds in Computational Complexity, which is conducted with support from the National Science Foundation. The first author was funded by the Simons Foundation, grant number 578919. The second author was funded by the Swedish Research Council (VR) grants 621-2012-5645 and 2016-00782 as well as by the Independent Research Fund Denmark grant 9040-00389B.

References

- [AB87] B. Alon and R. Boppana. The monotone circuit complexity of Boolean functions. *Combinatorica*, 7:1–22, 1987.
- [ABB16] James Aisenberg, María Luisa Bonet, and Sam Buss. Quasi-polynomial size Frege proofs of Frankl’s theorem on the trace of finite sets. *Journal of Symbolic Logic*, 81(2):1–24, 2016.
- [ABdR⁺18] Albert Atserias, Ilario Bonacina, Susanna F. de Rezende, Massimo Lauria, Jakob Nordström, and Alexander A. Razborov. Clique is hard on average for regular resolution. In *Proceedings of the 50th Annual ACM Symposium on Theory of Computing (STOC ’18)*, pages 866–877, June 2018.
- [ABE02] Albert Atserias, María Luisa Bonet, and Juan Luis Esteban. Lower bounds for the weak pigeonhole principle and random formulas beyond resolution. *Information and Computation*, 176(2):136–152, August 2002. Preliminary version in *ICALP ’01*.
- [ABLM08] Carlos Ansótegui, María Luisa Bonet, Jordi Levy, and Felip Manyà. Measuring the hardness of SAT instances. In *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI ’08)*, pages 222–228, July 2008.
- [ABSRW02] Michael Alekhovich, Eli Ben-Sasson, Alexander A. Razborov, and Avi Wigderson. Space complexity in propositional calculus. *SIAM Journal on Computing*, 31(4):1184–1211, April 2002. Preliminary version in *STOC ’00*.
- [AD08] Albert Atserias and Víctor Dalmau. A combinatorial characterization of resolution width. *Journal of Computer and System Sciences*, 74(3):323–334, May 2008. Preliminary version in *CCC ’03*.
- [AdRNV17] Joël Alwen, Susanna F. de Rezende, Jakob Nordström, and Marc Vinyals. Cumulative space in black-white pebbling and resolution. In *Proceedings of the 8th Innovations in Theoretical Computer Science Conference (ITCS ’17)*, volume 67 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:21, January 2017.
- [AFT11] Albert Atserias, Johannes Klaus Fichte, and Marc Thurley. Clause-learning algorithms with many restarts and bounded-width resolu-

- tion. *Journal of Artificial Intelligence Research*, 40:353–373, January 2011. Preliminary version in *SAT '09*.
- [AH19] Albert Atserias and Tuomas Hakoniemi. Size-degree trade-offs for Sums-of-Squares and Positivstellensatz proofs. In *Proceedings of the 34th Annual Computational Complexity Conference (CCC '19)*, volume 137 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:20, July 2019.
- [AHI05] Michael Alekhnovich, Edward A. Hirsch, and Dmitry Itsykson. Exponential lower bounds for the running time of DPLL algorithms on satisfiable formulas. *Journal of Automated Reasoning*, 35(1–3):51–72, October 2005. Preliminary version in *ICALP '04*.
- [AJPU07] Michael Alekhnovich, Jan Johannsen, Toniann Pitassi, and Alasdair Urquhart. An exponential separation between regular and general resolution. *Theory of Computing*, 3(5):81–102, May 2007. Preliminary version in *STOC '02*.
- [Ajt88] Miklós Ajtai. The complexity of the pigeonhole principle. In *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science (FOCS '88)*, pages 346–355, October 1988.
- [AKS10] Gilles Audemard, George Katsirelos, and Laurent Simon. A restriction of extended resolution for clause learning SAT solvers. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI '10)*, pages 15–20, July 2010.
- [AKV04] Albert Atserias, Phokion G. Kolaitis, and Moshe Y. Vardi. Constraint propagation as a proof system. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP '04)*, volume 3258 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2004.
- [Ale04] Michael Alekhnovich. Mutilated chessboard problem is exponentially hard for resolution. *Theoretical Computer Science*, 310(1–3):513–525, January 2004.
- [Ale11] Michael Alekhnovich. Lower bounds for k -DNF resolution on random 3-CNFs. *Computational Complexity*, 20(4):597–614, December 2011. Preliminary version in *STOC '05*.
- [ALN16] Albert Atserias, Massimo Lauria, and Jakob Nordström. Narrow proofs may be maximally long. *ACM Transactions on Computational Logic*, 17(3):19:1–19:30, May 2016. Preliminary version in *CCC '14*.
- [AM99] Dimitris Achlioptas and Michael Molloy. Almost all graphs with $2.522n$ edges are not 3-colorable. *Electronic Journal of Combinatorics*, 6:R29:1–R29:9, July 1999.
- [AM20] Albert Atserias and Moritz Müller. Automating resolution is np-hard. 67(5), 2020.
- [AMO15] Albert Atserias, Moritz Müller, and Sergi Oliva. Lower bounds for DNF-refutations of a relativized weak pigeonhole principle. *Journal of Symbolic Logic*, 80(2):450–476, June 2015. Preliminary version in *CCC '13*.
- [AO19] Albert Atserias and Joanna Ochremiak. Proof complexity meets

- algebra. *ACM Transactions on Computational Logic*, 20:1:1–1:46, February 2019. Preliminary version in *ICALP '17*.
- [AR03] Michael Alekhovich and Alexander A. Razborov. Lower bounds for polynomial calculus: Non-binomial case. *Proceedings of the Steklov Institute of Mathematics*, 242:18–35, 2003. Available at <http://people.cs.uchicago.edu/~razborov/files/misha.pdf>. Preliminary version in *FOCS '01*.
- [AR08] Michael Alekhovich and Alexander A. Razborov. Resolution is not automatizable unless $W[P]$ is tractable. *SIAM Journal on Computing*, 38(4):1347–1363, October 2008. Preliminary version in *FOCS '01*.
- [Ara00] Toshiyasu Arai. A bounded arithmetic AID for Frege systems. *Annals of Pure and Applied Logic*, 103:155–199, 2000.
- [AS09] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI '09)*, pages 399–404, July 2009.
- [AS12] Gilles Audemard and Laurent Simon. Refining restarts strategies for SAT and UNSAT. In *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (CP '12)*, volume 7514 of *Lecture Notes in Computer Science*, pages 118–126. Springer, October 2012.
- [AU00] Noriko H. Arai and Alasdair Urquhart. Local symmetries in propositional logic. In *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX '00)*, volume 1847 of *Lecture Notes in Computer Science*, pages 40–51. Springer, July 2000.
- [BB12] María Luisa Bonet and Samula R. Buss. An improved separation of regular resolution from pool resolution and clause learning. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT '12)*, volume 7317 of *Lecture Notes in Computer Science*, pages 44–57. Springer, June 2012.
- [BBG⁺17] Patrick Bennett, Ilario Bonacina, Nicola Galesi, Tony Huynh, Mike Molloy, and Paul Wollan. Space proof complexity for random 3-CNFs. *Information and Computation*, 255:165–176, 2017.
- [BBI16] Paul Beame, Chris Beck, and Russell Impagliazzo. Time-space tradeoffs in resolution: Superpolynomial lower bounds for superlinear space. *SIAM Journal on Computing*, 45(4):1612–1645, August 2016. Preliminary version in *STOC '12*.
- [BBI⁺18] María Luisa Bonet, Samula R. Buss, Alexey Ignatiev, João Marques-Silva, and António Morgado. MaxSAT resolution with the dual rail encoding. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI '18)*, pages 6565–6572, February 2018.
- [BBJ14] María Luisa Bonet, Sam Buss, and Jan Johannsen. Improved separations of regular resolution from clause learning proof systems.

- Journal of Artificial Intelligence Research*, 49:669–703, April 2014.
- [BBP95] Maria Luisa Bonet, Samuel R. Buss, and Toniann Pitassi. Are there hard examples for Frege systems? In P. Clote and J. Remmel, editors, *Feasible Mathematics II*, pages 30–56, Boston, 1995. Birkhäuser.
- [BC96] Samula R. Buss and Peter Clote. Cutting planes, connectivity and threshold logic. *Archive for Mathematical Logic*, 35:33–63, 1996.
- [BCE⁺98] Paul Beame, Stephen A. Cook, Jeff Edmonds, Russell Impagliazzo, and Toniann Pitassi. The relative complexity of NP search problems. *Journal of Computer and System Sciences*, 57(1):3–19, August 1998. Preliminary version in *STOC '95*.
- [BCMM05] Paul Beame, Joseph C. Culberson, David G. Mitchell, and Christopher Moore. The resolution complexity of random graph k -colorability. *Discrete Applied Mathematics*, 153(1-3):25–47, December 2005.
- [BD09] Michael Brickenstein and Alexander Dreyer. PolyBoRi: A framework for Gröbner-basis computations with Boolean polynomials. *Journal of Symbolic Computation*, 44(9):1326–1345, September 2009.
- [BDG⁺09] Michael Brickenstein, Alexander Dreyer, Gert-Martin Greuel, Markus Wedler, and Oliver Wienand. New developments in the theory of Gröbner bases and applications to formal verification. *Journal of Pure and Applied Algebra*, 213(8):1612–1635, August 2009.
- [BEGJ00] María Luisa Bonet, Juan Luis Esteban, Nicola Galesi, and Jan Johannsen. On the relative complexity of resolution refinements and cutting planes proof systems. *SIAM Journal on Computing*, 30(5):1462–1484, 2000. Preliminary version in *FOCS '98*.
- [Ber12] Christoph Berkholz. On the complexity of finding narrow proofs. In *Proceedings of the 53rd Annual IEEE Symposium on Foundations of Computer Science (FOCS '12)*, pages 351–360, October 2012.
- [Ber18] Christoph Berkholz. The relation between polynomial calculus, Sherali-Adams, and sum-of-squares proofs. In *Proceedings of the 35th Symposium on Theoretical Aspects of Computer Science (STACS '18)*, volume 96 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:14, February 2018.
- [BF15] Armin Biere and Andreas Fröhlich. Evaluating CDCL variable scoring schemes. In *Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing (SAT '15)*, volume 9340 of *Lecture Notes in Computer Science*, pages 405–422. Springer, September 2015.
- [BF19] Armin Biere and Andreas Fröhlich. Evaluating CDCL restart schemes. In *Proceedings of Pragmatics of SAT 2015 and 2018*, volume 59 of *EPIc Series in Computing*, pages 1–17, March 2019. Available at <https://easychair.org/publications/paper/RdBL>.
- [BFI⁺18] Paul Beame, Noah Fleming, Russell Impagliazzo, Antonina

- Kolokolova, Denis Pankratov, Toniann Pitassi, and Robert Robere. Stabbing planes. In *Proceedings of the 9th Innovations in Theoretical Computer Science Conference (ITCS '18)*, volume 94 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:20, January 2018.
- [BG01] María Luisa Bonet and Nicola Galesi. Optimality of size-width tradeoffs for resolution. *Computational Complexity*, 10(4):261–276, December 2001. Preliminary version in *FOCS '99*.
- [BG15] Ilario Bonacina and Nicola Galesi. A framework for space complexity in algebraic proof systems. *Journal of the ACM*, 62(3):23:1–23:20, June 2015. Preliminary version in *ITCS '13*.
- [BGIP01] Samula R. Buss, Dima Grigoriev, Russell Impagliazzo, and Toniann Pitassi. Linear gaps between degrees for the polynomial calculus modulo distinct primes. *Journal of Computer and System Sciences*, 62(2):267–289, March 2001. Preliminary version in *CCC '99*.
- [BGL13] Olaf Beyersdorff, Nicola Galesi, and Massimo Lauria. Parameterized complexity of DPLL search procedures. *ACM Transactions on Computational Logic*, 14(3):20:1–20:21, August 2013. Preliminary version in *SAT '11*.
- [BGT14] Ilario Bonacina, Nicola Galesi, and Neil Thapen. Total space in resolution. In *Proceedings of the 55th Annual IEEE Symposium on Foundations of Computer Science (FOCS '14)*, pages 641–650, October 2014.
- [BHJ08] Samula R. Buss, Jan Hoffmann, and Jan Johannsen. Resolution trees with lemmas: Resolution refinements that characterize DLL-algorithms with clause learning. *Logical Methods in Computer Science*, 4(4:13), December 2008.
- [BHP10] Paul Beame, Trinh Huynh, and Toniann Pitassi. Hardness amplification in proof complexity. In *Proceedings of the 42nd Annual ACM Symposium on Theory of Computing (STOC '10)*, pages 87–96, June 2010.
- [Bie06] Armin Biere. Tracecheck. <http://fmv.jku.at/tracecheck/>, 2006.
- [Bie08] Armin Biere. Adaptive restart strategies for conflict driven SAT solvers. In *Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing (SAT '08)*, volume 4996 of *Lecture Notes in Computer Science*, pages 28–33. Springer, May 2008.
- [BIK⁺94] Paul Beame, Russell Impagliazzo, Jan Krajíček, Toniann Pitassi, and Pavel Pudlák. Lower bounds on Hilbert's Nullstellensatz and propositional proofs. In *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science (FOCS '94)*, pages 794–806, November 1994.
- [BIK⁺96] Paul Beame, Russell Impagliazzo, Jan Krajíček, Toniann Pitassi, and Pavel Pudlák. Lower bounds on Hilbert's Nullstellensatz and propositional proofs. *Proceedings of the London Mathematical Society*, 73(3):1–26, 1996.

- [BIK⁺97] Samuel R. Buss, Russell Impagliazzo, Jan Krajíček, Pavel Pudlák, Alexander A. Razborov, and Jiri Sgall. Proof complexity in algebraic systems and bounded depth Frege systems with modular counting. *Computational Complexity*, 6(3):256–298, 1997.
- [BIKS18] Samula R. Buss, Dmitry Itsykson, Alexander Knop, and Dmitry Sokolov. Reordering rule makes OBDD proof systems stronger. In *Proceedings of the 33rd Annual Computational Complexity Conference (CCC '18)*, volume 102 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:24, June 2018.
- [BIS07] Paul Beame, Russell Impagliazzo, and Ashish Sabharwal. The resolution complexity of independent sets and vertex covers in random graphs. *Computational Complexity*, 16(3):245–297, October 2007. Preliminary version in *CCC '01*.
- [BK14] Samula R. Buss and Leszek Koodziejczyk. Small stone in pool. *Logical Methods in Computer Science*, 10(2):16:1–16:22, June 2014.
- [BKS04] Paul Beame, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, December 2004. Preliminary version in *IJCAI '03*.
- [Bla37] Archie Blake. *Canonical Expressions in Boolean Algebra*. PhD thesis, University of Chicago, 1937.
- [BLLM14] Armin Biere, Daniel Le Berre, Emmanuel Lonca, and Norbert Manthey. Detecting cardinality constraints in CNF. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, volume 8561 of *Lecture Notes in Computer Science*, pages 285–301. Springer, July 2014.
- [BN20] Christoph Berkholz and Jakob Nordström. Supercritical space-width trade-offs for resolution. *SIAM Journal on Computing*, 49(1):98–118, February 2020. Preliminary version in *ICALP '16*.
- [BNT13] Chris Beck, Jakob Nordström, and Bangsheng Tang. Some trade-off results for polynomial calculus. In *Proceedings of the 45th Annual ACM Symposium on Theory of Computing (STOC '13)*, pages 813–822, May 2013.
- [BOCIP02] Joshua Buresh-Oppenheimer, Matthew Clegg, Russell Impagliazzo, and Toniann Pitassi. Homogenization and the polynomial calculus. *Computational Complexity*, 11(3-4):91–108, 2002. Preliminary version in *ICALP '00*.
- [Bon16] Ilario Bonacina. Total space in resolution is at least width squared. In *Proceedings of the 43rd International Colloquium on Automata, Languages and Programming (ICALP '16)*, volume 55 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 56:1–56:13, July 2016.
- [BP96a] Paul Beame and Toniann Pitassi. An exponential separation between the parity principle and the pigeonhole principle. *Annals of Pure and Applied Logic*, 80(3):195–228, 1996.
- [BP96b] Paul Beame and Toniann Pitassi. Simplified and improved resolution lower bounds. In *Proceedings of the 37th Annual IEEE Sym-*

- posium on Foundations of Computer Science (FOCS '96)*, pages 274–282, October 1996.
- [BP98a] Paul Beame and Toniann Pitassi. Propositional proof complexity: Past, present, and future. *Bulletin of the European Association for Theoretical Computer Science*, 65:66–89, June 1998.
- [BP98b] Samula R. Buss and Toniann Pitassi. Good degree bounds on Nullstellensatz refutations of the induction principle. *Journal of Computer and System Sciences*, 2(57):162–171, October 1998. Preliminary version in *CCC '96*.
- [BPR00] María Luisa Bonet, Toniann Pitassi, and Ran Raz. On interpolation and automatization for Frege systems. *SIAM Journal on Computing*, 29(6):1939–1967, 2000.
- [BPT14] Arnold Beckmann, Pavel Pudlák, and Neil Thapen. Parity games and propositional proofs. *ACM Transactions on Computational Logic*, 15(2):17:1–30, 2014.
- [Bry92] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagram. *ACM Computing Surveys*, 24(3):293–318, 1992.
- [BS97] Roberto J. Bayardo Jr. and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI '97)*, pages 203–208, July 1997.
- [BS09] Eli Ben-Sasson. Size-space tradeoffs for resolution. *SIAM Journal on Computing*, 38(6):2511–2525, May 2009. Preliminary version in *STOC '02*.
- [BS14] Boaz Barak and David Steurer. Sum-of-squares proofs and the quest toward optimal algorithms. In *Proceedings of the International Congress of Mathematicians (ICM)*, volume IV, pages 509–533, August 2014. Available at http://www.icm2014.org/download/Proceedings_Volume_IV.pdf.
- [BSG03] Eli Ben-Sasson and Nicola Galesi. Space complexity of random formulae in resolution. *Random Structures and Algorithms*, 23(1):92–109, August 2003. Preliminary version in *CCC '01*.
- [BSI99] Eli Ben-Sasson and Russell Impagliazzo. Random CNF's are hard for the polynomial calculus. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science (FOCS '99)*, pages 415–421, October 1999. Journal version in [BSI10].
- [BSI10] Eli Ben-Sasson and Russell Impagliazzo. Random CNF's are hard for the polynomial calculus. *Computational Complexity*, 19(4):501–519, 2010. Preliminary version in *FOCS '99*.
- [BSIW04] Eli Ben-Sasson, Russell Impagliazzo, and Avi Wigderson. Near optimal separation of tree-like and general resolution. *Combinatorica*, 24(4):585–603, September 2004.
- [BSJ10] Eli Ben-Sasson and Jan Johannsen. Lower bounds for width-restricted clause learning on small width formulas. In *Proceedings of the 13th International Conference on Theory and Applications*

- of *Satisfiability Testing (SAT '10)*, volume 6175 of *Lecture Notes in Computer Science*, pages 16–29. Springer, July 2010.
- [BSN08] Eli Ben-Sasson and Jakob Nordström. Short proofs may be spacious: An optimal separation of space and length in resolution. In *Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS '08)*, pages 709–718, October 2008.
- [BSN11] Eli Ben-Sasson and Jakob Nordström. Understanding space in proof complexity: Separations and trade-offs via substitutions. In *Proceedings of the 2nd Symposium on Innovations in Computer Science (ICS '11)*, pages 401–416, January 2011.
- [BSW01] Eli Ben-Sasson and Avi Wigderson. Short proofs are narrow—resolution made simple. *Journal of the ACM*, 48(2):149–169, March 2001. Preliminary version in *STOC '99*.
- [BT88] Samuel R. Buss and György Turán. Resolution proofs of generalized pigeonhole principles. *Theoretical Computer Science*, 62:311–317, 1988.
- [BT19] Samula R. Buss and Neil Thapen. DRAT proofs, propagation redundancy, and extended resolution. In *Proceedings of the 22nd International Conference on Theory and Applications of Satisfiability Testing (SAT '19)*, volume 11628 of *Lecture Notes in Computer Science*, pages 71–89. Springer, July 2019.
- [Bus86] Samuel R. Buss. *Bounded Arithmetic*. Bibliopolis, Naples, 1986. Revision of PhD thesis.
- [Bus87] Samuel R. Buss. Polynomial size proofs of the propositional pigeonhole principle. *Journal of Symbolic Logic*, 52:916–927, 1987.
- [Bus98] Samula R. Buss. Lower bounds on Nullstellensatz proofs via designs. In *Proof Complexity and Feasible Arithmetics*, volume 39 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 59–71. American Mathematical Society, 1998. Available at <http://www.math.ucsd.edu/~sbuss/ResearchWeb/designs/>.
- [Bus99] Samuel R. Buss. Propositional proof complexity: An introduction. In U. Berger and H. Schwichtenberg, editors, *Computational Logic*, pages 127–178. Springer-Verlag, Berlin, 1999.
- [Bus12] Samuel R. Buss. Towards NP-P via proof complexity and proof search. *Annals of Pure and Applied Logic*, 163(9):1163–1182, 2012.
- [Bus15] Samula R. Buss. Quasipolynomial size proofs of the propositional pigeonhole principle. *Theoretical Computer Science*, 576:77–84, April 2015.
- [CaD] CaDiCaL. <http://fmv.jku.at/cadical/>.
- [CCT87] William Cook, Collette Rene Coullard, and György Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, November 1987.
- [CEI96] Matthew Clegg, Jeffery Edmonds, and Russell Impagliazzo. Using the Groebner basis algorithm to find proofs of unsatisfiability. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing (STOC '96)*, pages 174–183, May 1996.

- [Chv73] Vašek Chvátal. Edmonds polytopes and a hierarchy of combinatorial problems. *Discrete Mathematics*, 4(1):305–337, 1973.
- [CIP09] Chris Calabro, Russell Impagliazzo, and Ramamohan Paturi. The complexity of satisfiability of small depth circuits. In *Revised Selected Papers from the 4th International Workshop on Parameterized and Exact Computation (IWPEC '09)*, volume 5917 of *Lecture Notes in Computer Science*, pages 75–85. Springer, September 2009.
- [CK05] Donald Chai and Andreas Kuehlmann. A fast pseudo-Boolean constraint solver. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(3):305–317, March 2005. Preliminary version in *DAC '03*.
- [CM05] Stephen A. Cook and Tsuyoshi Morioka. Quantified propositional calculus and a second-order theory for NC^1 . *Archive for Mathematical Logic*, 44:711–749, 2005.
- [CN10] Stephen A. Cook and Phuong Nguyen. *Logical Foundations of Proof Complexity*. Cambridge University Press, July 2010.
- [CNF] CNFgen: Combinatorial benchmarks for SAT solvers. <https://massimolauria.net/cnfgen/>.
- [Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC '71)*, pages 151–158, May 1971.
- [Coo75] Stephen A. Cook. Feasibly constructive proofs and the propositional calculus (preliminary version). In *Proceedings of the 7th Annual ACM Symposium on Theory of Computing (STOC '75)*, pages 83–97, May 1975.
- [Coo88] Stephen A. Cook. Short propositional formulas represent nondeterministic computations. *Information Processing Letters*, 26:269–270, 1988.
- [CPL] IBM ILOG CPLEX optimization studio. <https://www.ibm.com/products/ilog-cplex-optimization-studio>.
- [CR79] Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic*, 44(1):36–50, March 1979. Preliminary version in *STOC '74*.
- [Cry] CryptoMiniSat. <https://github.com/msoos/cryptominisat/>.
- [CS80] David A. Carlson and John E. Savage. Graph pebbling with many free pebbles can be difficult. In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing (STOC '80)*, pages 326–332, 1980.
- [CS82] David A. Carlson and John E. Savage. Extreme time-space trade-offs for graphs with small space requirements. *Information Processing Letters*, 14(5):223–227, 1982.
- [CS88] Vašek Chvátal and Endre Szemerédi. Many hard examples for resolution. *Journal of the ACM*, 35(4):759–768, October 1988.
- [CT12] Eden Chlamtáč and Madhur Tulsiani. Convex relaxations and integrality gaps. In Miguel F. Anjos and Jean B. Lasserre, editors, *Handbook on Semidefinite, Conic and Polynomial Optimization*,

pages 139–169. Springer, 2012.

- [DBB17] Jo Devriendt, Bart Bogaerts, and Maurice Bruynooghe. Symmetric explanation learning: Effective dynamic symmetry handling for SAT. In *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT '17)*, volume 10491 of *Lecture Notes in Computer Science*, pages 83–100. Springer, August 2017.
- [DBBD16] Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe, and Marc De-necker. Improved static symmetry breaking for SAT. In *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing (SAT '16)*, volume 9710 of *Lecture Notes in Computer Science*, pages 104–122. Springer, July 2016.
- [DBM00] Olivier Dubois, Yacine Boufkhad, and Jacques Mandler. Typical random 3-SAT formulae and the satisfiability threshold. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '00)*, pages 126–127, January 2000.
- [Dev20] Jo Devriendt. Watched propagation of 0-1 integer linear constraints. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP '20)*, volume 12333 of *Lecture Notes in Computer Science*, pages 160–176. Springer, September 2020.
- [DG02] Heidi E. Dixon and Matthew L. Ginsberg. Inference methods for a pseudo-Boolean satisfiability solver. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI '02)*, pages 635–640, July 2002.
- [DGN20] Jo Devriendt, Ambros Gleixner, and Jakob Nordström. Learn to relax: Integrating 0-1 integer linear programming with pseudo-Boolean conflict-driven search. In *Proceedings of the 17th International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR '20)*, pages xxiv–xxv, September 2020.
- [Dix04] Heidi E. Dixon. *Automating Pseudo-Boolean Inference within a DPLL Framework*. PhD thesis, University of Oregon, 2004. Available at <http://www.cirl.uoregon.edu/dixon/papers/dixonDissertation.pdf>.
- [DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.
- [DLLMM08] Jesús A. De Loera, Jon Lee, Peter N. Malkin, and Susan Margulies. Hilbert’s Nullstellensatz and an algorithm for proving combinatorial infeasibility. In *Proceedings of the 21st International Symposium on Symbolic and Algebraic Computation (ISSAC '08)*, pages 197–206, July 2008.
- [DLLMM11] Jesús A. De Loera, Jon Lee, Peter N. Malkin, and Susan Margulies. Computing infeasibility certificates for combinatorial problems through Hilbert’s Nullstellensatz. *Journal of Symbolic Computation*, 46(11):1260–1283, November 2011.

- [DLLMO09a] Jesús A. De Loera, Jon Lee, Susan Margulies, and Shmuel Onn. Expressing combinatorial problems by systems of polynomial equations and Hilbert’s Nullstellensatz. *Combinatorics, Probability and Computing*, 18(4):551–582, July 2009.
- [DLLMO09b] Jesús A. De Loera, Jon Lee, Susan Margulies, and Shmuel Onn. Expressing combinatorial problems by systems of polynomial equations and Hilbert’s Nullstellensatz. *Combinatorics, Probability and Computing*, 18(04):551–582, July 2009.
- [DMR09] Stefan S. Dantchev, Barnaby Martin, and Martin Rhodes. Tight rank lower bounds for the Sherali–Adams proof system. *Theoretical Computer Science*, 410(21–23):2054–2063, May 2009.
- [DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.
- [DR01] Stefan S. Dantchev and Søren Riis. “Planar” tautologies hard for resolution. In *Proceedings of the 42nd Annual IEEE Symposium on Foundations of Computer Science (FOCS ’01)*, pages 220–229, October 2001.
- [dRGN⁺20] Susanna F. de Rezende, Mika Göös, Jakob Nordström, Toniann Pitassi, Robert Robere, and Dmitry Sokolov. Automating algebraic proof systems is NP-hard. Technical Report TR20-064, Electronic Colloquium on Computational Complexity (ECCC), May 2020.
- [dRLM⁺20] Susanna F. de Rezende, Massimo Lauria, Or Meir, Jakob Nordström, and Dmitry Sokolov. Manuscript in preparation, 2020.
- [dRMN⁺20] Susanna F. de Rezende, Or Meir, Jakob Nordström, Toniann Pitassi, Robert Robere, and Marc Vinyals. Lifting with simple gadgets and applications to circuit and proof complexity. In *Proceedings of the 61st Annual IEEE Symposium on Foundations of Computer Science (FOCS ’20)*, November 2020. To appear.
- [dRNMR19] Susanna F. de Rezende, Jakob Nordström, Or Meir, and Robert Robere. Nullstellensatz size-degree trade-offs from reversible pebbling. In *Proceedings of the 34th Annual Computational Complexity Conference (CCC ’19)*, volume 137 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:16, July 2019.
- [dRNV16] Susanna F. de Rezende, Jakob Nordström, and Marc Vinyals. How limited interaction hinders real communication (and what it means for proof and circuit complexity). In *Proceedings of the 57th Annual IEEE Symposium on Foundations of Computer Science (FOCS ’16)*, pages 295–304, October 2016.
- [DT20] Daniel Dadush and Samarth Tiwari. On the complexity of branching proofs. In *Proceedings of the 35th Annual Computational Complexity Conference (CCC ’20)*, volume 169 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 34:1–34:35, July 2020.
- [EGCG⁺18] Jan Elffers, Jesús Giráldez-Cru, Stephan Gocht, Jakob Nordström, and Laurent Simon. Seeking practical CDCL insights from theoretical SAT benchmarks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI ’18)*, pages 1300–1308, July 2018.

- [EGCNV18] Jan Elffers, Jesús Giráldez-Cru, Jakob Nordström, and Marc Vinyals. Using combinatorial benchmarks to probe the reasoning power of pseudo-Boolean solvers. In *Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing (SAT '18)*, volume 10929 of *Lecture Notes in Computer Science*, pages 75–93. Springer, July 2018.
- [EGMN20] Jan Elffers, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Justifying all differences using pseudo-Boolean reasoning. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI '20)*, pages 1486–1494, February 2020.
- [EJL⁺16] Jan Elffers, Jan Johannsen, Massimo Lauria, Thomas Magnard, Jakob Nordström, and Marc Vinyals. Trade-offs between time and memory in a tighter model of CDCL SAT solvers. In *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing (SAT '16)*, volume 9710 of *Lecture Notes in Computer Science*, pages 160–176. Springer, July 2016.
- [EN18] Jan Elffers and Jakob Nordström. Divide and conquer: Towards faster pseudo-Boolean solving. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI '18)*, pages 1291–1299, July 2018.
- [EN20] Jan Elffers and Jakob Nordström. A cardinal improvement to pseudo-Boolean solving. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI '20)*, pages 1495–1503, February 2020.
- [ES04] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *6th International Conference on Theory and Applications of Satisfiability Testing (SAT '03), Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004.
- [ES06] Niklas Eén and Niklas Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, March 2006.
- [ET01] Juan Luis Esteban and Jacobo Torán. Space bounds for resolution. *Information and Computation*, 171(1):84–97, 2001. Preliminary versions of these results appeared in *STACS '99* and *CSL '99*.
- [FKP19] Noah Fleming, Pravesh Kothari, and Toniann Pitassi. Semialgebraic proofs and efficient algorithm design. *Foundations and Trends in Theoretical Computer Science*, 14(1–2):1–221, December 2019.
- [FLM⁺13] Yuval Filmus, Massimo Lauria, Mladen Mikša, Jakob Nordström, and Marc Vinyals. Towards an understanding of polynomial calculus: New separations and lower bounds (Extended abstract). In *Proceedings of the 40th International Colloquium on Automata, Languages and Programming (ICALP '13)*, volume 7965 of *Lecture Notes in Computer Science*, pages 437–448. Springer, July 2013.
- [FLM⁺15] Yuval Filmus, Massimo Lauria, Mladen Mikša, Jakob Nordström, and Marc Vinyals. From small space to small width in resolution. *ACM Transactions on Computational Logic*, 16(4):28:1–28:15, July 2015. Preliminary version in *STACS '14*.

- [FLN⁺15] Yuval Filmus, Massimo Lauria, Jakob Nordström, Noga Ron-Zewi, and Neil Thapen. Space complexity in polynomial calculus. *SIAM Journal on Computing*, 44(4):1119–1153, August 2015. Preliminary version in *CCC '12*.
- [FPPR17] Noah Fleming, Denis Pankratov, Toniann Pitassi, and Robert Robere. Random $\Theta(\log n)$ -CNFs are hard for cutting planes. In *Proceedings of the 58th Annual IEEE Symposium on Foundations of Computer Science (FOCS '17)*, pages 109–120, October 2017.
- [Gen35] Gerhard Gentzen. Untersuchungen über das logische Schliessen I & II. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in [Gen69], pp. 68–131.
- [Gen69] Gerhard Gentzen. *Collected Papers of Gerhard Gentzen*. North-Holland, 1969. Edited by M. E. Szabo.
- [GGKS18] Ankit Garg, Mika Göös, Pritish Kamath, and Dmitry Sokolov. Monotone circuit lower bounds from resolution. In *Proceedings of the 50th Annual ACM Symposium on Theory of Computing (STOC '18)*, pages 902–911, June 2018.
- [GHP02a] Dima Grigoriev, Edward A. Hirsch, and Dmitrii V. Pasechnik. Complexity of semialgebraic proofs. *Moscow Mathematical Journal*, 2(4):647–679, 2002. Preliminary version in *STACS '02*.
- [GHP02b] Dima Grigoriev, Edward A. Hirsch, and Dmitrii V. Pasechnik. Exponential lower bound for static semi-algebraic proofs. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming (ICALP '02)*, volume 2380 of *Lecture Notes in Computer Science*, pages 257–268. Springer, July 2002.
- [GKKS09] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. On the implementation of weight constraint rules in conflict-driven ASP solvers. In *Proceedings of the 25th International Conference on Logic Programming (ICLP '09)*, volume 5649 of *Lecture Notes in Computer Science*, pages 250–264. Springer, July 2009.
- [GKMP20] Mika Göös, Sajin Korothe, Ian Mertz, and Toniann Pitassi. Automating cutting planes is NP-hard. In *Proceedings of the 52nd Annual ACM Symposium on Theory of Computing (STOC '20)*, pages 68–77, June 2020.
- [GKRS19] Mika Göös, Pritish Kamath, Robert Robere, and Dmitry Sokolov. Adventures in monotone complexity and TFNP. In *Proceedings of the 10th Innovations in Theoretical Computer Science Conference (ITCS '19)*, volume 124 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:19, January 2019.
- [GKT19] Nicola Galesi, Leszek Koodziejczyk, and Neil Thapen. Polynomial calculus space and resolution width. In *Proceedings of the 60th Annual IEEE Symposium on Foundations of Computer Science (FOCS '19)*, pages 1325–1337, November 2019.
- [GL10a] Nicola Galesi and Massimo Lauria. On the automatizability of polynomial calculus. *Theory of Computing Systems*, 47(2):491–506, August 2010.

- [GL10b] Nicola Galesi and Massimo Lauria. Optimality of size-degree trade-offs for polynomial calculus. *ACM Transactions on Computational Logic*, 12(1):4:1–4:22, November 2010.
- [GMM⁺20] Stephan Gocht, Ross McBride, Ciaran McCreesh, Jakob Nordström, Patrick Prosser, and James Trimble. Certifying solvers for clique and maximum common (connected) subgraph problems. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP '20)*, volume 12333 of *Lecture Notes in Computer Science*, pages 338–357. Springer, September 2020.
- [GMN20a] Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Subgraph isomorphism meets cutting planes: Solving with certified solutions. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI '20)*, pages 1134–1140, July 2020.
- [GMN20b] Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. VeriPB: The easy way to make your combinatorial search algorithm trustworthy. Presented at the workshop *From Constraint Programming to Trustworthy AI* at the *26th International Conference on Principles and Practice of Constraint Programming (CP '20)*. Paper available at http://www.cs.ucc.ie/~bg6/cptai/2020/papers/CPTAI_2020_paper_2.pdf, September 2020.
- [GN03] Evgueni Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '03)*, pages 886–891, March 2003.
- [GNY19] Stephan Gocht, Jakob Nordström, and Amir Yehudayoff. On division versus saturation in pseudo-Boolean solving. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI '19)*, pages 1711–1718, August 2019.
- [Goc17] Stephan Gocht. Personal communication, 2017.
- [Goe90] Andreas Goerdt. Cutting plane versus Frege proof systems. In *Proceedings of the 4th International Workshop on Computer Science Logic (CSL '90)*, volume 533 of *Lecture Notes in Computer Science*, pages 174–194. Springer, October 1990.
- [Gom63] Ralph E. Gomory. An algorithm for integer solutions of linear programs. In R. L. Graves and P. Wolfe, editors, *Recent Advances in Mathematical Programming*, pages 269–302. McGraw-Hill, New York, 1963.
- [GP18a] Mika Göös and Toniann Pitassi. Communication lower bounds via critical block sensitivity. *SIAM Journal on Computing*, 47(5):1778–1806, October 2018. Preliminary version in *STOC '14*.
- [GP18b] Joshua A. Grochow and Toniann Pitassi. Circuit complexity, proof complexity, and polynomial identity testing: The ideal proof system. *Journal of the ACM*, 65(6):37:1–37:59, November 2018. Preliminary version in *FOCS '14*.
- [GPT15] Nicola Galesi, Pavel Pudlák, and Neil Thapen. The space complex-

- ity of cutting planes refutations. In *Proceedings of the 30th Annual Computational Complexity Conference (CCC '15)*, volume 33 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 433–447, June 2015.
- [GT78] John R. Gilbert and Robert Endre Tarjan. Variations of a pebble game on graphs. Technical Report STAN-CS-78-661, Stanford University, 1978. Available at <http://infolab.stanford.edu/TR/CS-TR-78-661.html>.
- [Gur] Gurobi optimizer. <https://www.gurobi.com/>.
- [GV01] Dima Grigoriev and Nicolai Vorobjov. Complexity of Null- and Positivstellensatz proofs. *Annals of Pure and Applied Logic*, 113(1–3):153–160, December 2001.
- [GZ03] Jan Frisco Groote and Hans Zantema. Resolution and binary decision diagrams cannot simulate each other polynomially. *Discrete Applied Mathematics*, 130(2):157–171, 2003.
- [Hak85] Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39(2-3):297–308, August 1985.
- [Hås86] Johan Håstad. Almost optimal lower bounds for small depth circuits. In *Proceedings of the 18-th Annual ACM Symposium on Theory of Computing (STOC)*, pages 6–20, 1986.
- [Hås17] Johan Håstad. On small-depth frege proofs for Tseitin for grids. In *Proceedings of the 58th Annual IEEE Symposium on Foundations of Computer Science (FOCS '17)*, pages 97–108, October 2017.
- [HB18] Marijn J. H. Heule and Armin Biere. What a difference a variable makes. In *Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 18)*, volume 10806 of *Lecture Notes in Computer Science*, pages 75–92. Springer, April 2018.
- [HBPV08] Philipp Hertel, Fahiem Bacchus, Toniann Pitassi, and Allen Van Gelder. Clause learning can effectively P-simulate general propositional resolution. In *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI '08)*, pages 283–290, July 2008.
- [HC99] Armin Haken and Stephen A. Cook. An exponential lower bound for the size of monotone real circuits. *Journal of Computer and System Sciences*, 58(2):326–335, April 1999.
- [HHW13a] Marijn J. H. Heule, Warren A. Hunt Jr. , and Nathan Wetzler. Trimming while checking clausal proofs. In *Proceedings of the 13th International Conference on Formal Methods in Computer-Aided Design (FMCAD '13)*, pages 181–188, October 2013.
- [HHW13b] Marijn J. H. Heule, Warren A. Hunt Jr. , and Nathan Wetzler. Verifying refutations with extended resolution. In *Proceedings of the 24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *Lecture Notes in Computer Science*, pages 345–359. Springer, June 2013.
- [HHW15] Marijn J. H. Heule, Warren A. Hunt Jr. , and Nathan Wetzler. Expressing symmetry breaking in DRAT proofs. In *Proceedings of the 25th International Conference on Automated Deduction*

- (CADE-25), volume 9195 of *Lecture Notes in Computer Science*, pages 591–606. Springer, August 2015.
- [HJB10] Marijn Heule, Matti Järvisalo, and Armin Biere. Clause elimination procedures for CNF formulas. In *Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR '10)*, volume 6397 of *Lecture Notes in Computer Science*, pages 357–371. Springer, October 2010.
- [HKB17] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Short proofs without new variables. In *Proceedings of the 26th International Conference on Automated Deduction (CADE-26)*, volume 10395 of *Lecture Notes in Computer Science*, pages 130–147. Springer, August 2017.
- [HKB19] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Encoding redundancy for satisfaction-driven clause learning. In *Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 19)*, volume 11427 of *Lecture Notes in Computer Science*, pages 41–58. Springer, April 2019.
- [HKB20] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Strong extension-free proof systems. *Journal of Automated Reasoning*, 64(3):533–554, 2020. Extended version of [HKB17].
- [HKSB17] Marijn J. H. Heule, Benjamin Kiesl, Martina Seidl, and Armin Biere. PRuning through satisfaction. In *13th International Haifa Verification Conference (HVC '17)*, volume 10629 of *Lecture Notes in Computer Science*, pages 179–194. Springer, November 2017.
- [HLW06] Shlomo Hoory, Nathan Linial, and Avi Wigderson. Expander graphs and their applications. *Bulletin of the American Mathematical Society*, 43(4):439–561, October 2006.
- [HN12] Trinh Huynh and Jakob Nordström. On the virtue of succinct proofs: Amplifying communication complexity hardness to time-space trade-offs in proof complexity (Extended abstract). In *Proceedings of the 44th Annual ACM Symposium on Theory of Computing (STOC '12)*, pages 233–248, May 2012.
- [Hoo88] John N. Hooker. Generalized resolution and cutting planes. *Annals of Operations Research*, 12(1):217–239, December 1988.
- [Hoo92] John N. Hooker. Generalized resolution for 0-1 linear inequalities. *Annals of Mathematics and Artificial Intelligence*, 6(1):271–286, March 1992.
- [HP93] Petr Hájek and Pavel Pudlák. *Metamathematics of First-order Arithmetic*. Perspectives in Mathematical Logic. Springer-Verlag, Berlin, 1993.
- [HP17] Pavel Hrubeš and Pavel Pudlák. Random formulas, monotone circuits, and interpolation. In *Proceedings of the 58th Annual IEEE Symposium on Foundations of Computer Science (FOCS '17)*, pages 121–131, October 2017.
- [HPV77] John Hopcroft, Wolfgang J. Paul, and Leslie Valiant. On time versus space. *Journal of the ACM*, 24(2):332–337, April 1977. Pre-

- liminary version in *FOCS '75*.
- [HS09] Hyojung Han and Fabio Somenzi. On-the-fly clause improvement. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT '09)*, volume 5584 of *Lecture Notes in Computer Science*, pages 209–222. Springer, July 2009.
- [HT15] Pavel Hrubeš and Iddo Zameret. Short proofs for the determinant identities. *SIAM Journal on Computing*, 44(2):340–383, April 2015. Preliminary version in *STOC '12*.
- [Hua07] Jinbo Huang. The effect of restarts on the efficiency of clause learning. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI '07)*, pages 2318–2323, January 2007.
- [Hua10] Jinbo Huang. Extended clause learning. *Artificial Intelligence*, 174(15):1277–1284, October 2010.
- [HvM05] Marijn J. H. Heule and Hans van Maaren. Aligning CNF- and equivalence-reasoning. In *7th International Conference on Theory and Applications of Satisfiability Testing (SAT '04), Selected Revised Papers*, volume 3542 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 2005.
- [HvM06] Marijn J. H. Heule and Hans van Maaren. March_dl: Adding adaptive heuristics and a new branching strategy. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:47–59, 2006.
- [IKRS17] Dmitry Itsykson, Alexander Knop, Andrei E. Romashchenko, and Dmitry Sokolov. On OBDD-based algorithms and proof systems that dynamically change order of variables. In *Proceedings of the 34th Symposium on Theoretical Aspects of Computer Science (STACS '17)*, volume 66 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 43:1–43:14, March 2017.
- [IMMS17] Alexey Ignatiev, António Morgado, and João Marques-Silva. On tackling the limits of resolution in SAT solving. In *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT '17)*, volume 10491 of *Lecture Notes in Computer Science*, pages 164–183. Springer, August 2017.
- [IP01] Russell Impagliazzo and Ramamohan Paturi. On the complexity of k -SAT. *Journal of Computer and System Sciences*, 62(2):367–375, March 2001. Preliminary version in *CCC '99*.
- [IPS99] Russell Impagliazzo, Pavel Pudlák, and Jiří Sgall. Lower bounds for the polynomial calculus and the Gröbner basis algorithm. *Computational Complexity*, 8(2):127–144, 1999.
- [JĪ1] Matti Järvisalo. On the relative efficiency of DPLL and OBDDs with axiom and join. In *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP '11)*, volume 6876 of *Lecture Notes in Computer Science*, pages 429–437. Springer, September 2011.
- [JdM13] Dejan Jovanovic and Leonardo de Moura. Cutting to the chase solving linear integer arithmetic. *Journal of Automated Reasoning*,

- 51(1):79–108, June 2013. Preliminary version in *CADE-23* 2011.
- [JHB12] Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. Inprocessing rules. In *Proceedings of the 6th International Joint Conference on Automated Reasoning (IJCAR '12)*, volume 7364 of *Lecture Notes in Computer Science*, pages 355–370. Springer, June 2012.
- [JMNv12] Matti Järvisalo, Arie Matsliah, Jakob Nordström, and Stanislav Živný. Relating proof complexity measures and practical hardness of SAT. In *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (CP '12)*, volume 7514 of *Lecture Notes in Computer Science*, pages 316–331. Springer, October 2012.
- [KBK19] Daniela Kaufmann, Armin Biere, and Manuel Kauers. Incremental column-wise verification of arithmetic circuits using computer algebra. *Formal Methods in Systems Design*, February 2019.
- [KI02] Jan Krajíček and Russell Impagliazzo. A note on conservativity relations among bounded arithmetic theories. *Mathematical Logic Quarterly*, 48(3):375–377, 2002.
- [KI06] Arist Kojevnikov and Dmitry Itsykson. Lower bounds of static Lovász–Schrijver calculus proofs for Tseitin tautologies. In *Proceedings of the 33rd International Colloquium on Automata, Languages and Programming (ICALP '06)*, volume 4051 of *Lecture Notes in Computer Science*, pages 323–334. Springer, July 2006.
- [KN97] Eyal Kushilevitz and Noam Nisan. *Communication Complexity*. Cambridge University Press, 1997.
- [KN20] Janne I. Kokkala and Jakob Nordström. Using resolution proofs to analyse CDCL solvers. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP '20)*, volume 12333 of *Lecture Notes in Computer Science*, pages 427–444. Springer, September 2020.
- [KPW95] Jan Krajíček, Pavel Pudlák, and Alan R. Woods. An exponential lower bound to the size of bounded depth Frege proofs of the pigeonhole principle. *Random Structures and Algorithms*, 7(1):15–40, 1995. Preliminary version in *STOC '92*.
- [Kra94] Jan Krajíček. Lower bounds to the size of constant-depth propositional proofs. *Journal of Symbolic Logic*, 59:73–86, 1994.
- [Kra95] Jan Krajíček. *Bounded Arithmetic, Propositional Logic and Complexity Theory*, volume 60 of *Encyclopedia of Mathematics and Its Applications*. Cambridge University Press, November 1995.
- [Kra97] Jan Krajíček. Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. *Journal of Symbolic Logic*, 62(2):457–486, June 1997.
- [Kra08] Jan Krajíček. An exponential lower bound for a constraint propagation proof system based on ordered binary decision diagrams. *Journal of Symbolic Logic*, 73(1):227–237, 2008.
- [Kra19] Jan Krajíček. *Proof Complexity*, volume 170 of *Encyclopedia of Mathematics and Its Applications*. Cambridge University Press, March 2019.

- [Kri85] Balakrishnan Krishnamurthy. Short proofs for tricky formulas. *Acta Informatica*, 22(3):253–275, August 1985.
- [KRPH18] Benjamin Kiesl, Adrián Rebola-Pardo, and Marijn J. H. Heule. Extended resolution simulates DRAT. In *Proceedings of the 9th International Joint Conference on Automated Reasoning (IJCAR '18)*, volume 10900 of *Lecture Notes in Computer Science*, pages 516–531. Springer, July 2018.
- [KSM11] Hadi Katebi, Karem A. Sakallah, and João P. Marques-Silva. Empirical study of the anatomy of modern SAT solvers. In *Proceedings of the 14th International Conference on Theory and Applications of Satisfiability Testing (SAT '11)*, volume 6695 of *Lecture Notes in Computer Science*, pages 343–356. Springer, June 2011.
- [Las01a] Jean B. Lasserre. An explicit exact SDP relaxation for nonlinear 0-1 programs. In *Proceedings of the 8th International Conference on Integer Programming and Combinatorial Optimization (IPCO '01)*, volume 2081 of *Lecture Notes in Computer Science*, pages 293–303. Springer, June 2001.
- [Las01b] Jean B. Lasserre. Global optimization with polynomials and the problem of moments. *SIAM Journal of Optimization*, 11(3):796–817, 2001.
- [Lau01] Monique Laurent. A comparison of the Sherali–Adams, Lovász–Schrijver and Lasserre relaxations for 0-1 programming. *Mathematics of Operations Research*, 28:470–496, 2001.
- [LBD⁺20] Vincent Liew, Paul Beame, Jo Devriendt, Jan Elffers, and Jakob Nordström. Verifying properties of bit-vector multiplication using cutting planes reasoning. In *Proceedings of the 20th Conference on Formal Methods in Computer-Aided Design (FMCAD '20)*, pages 194–204, September 2020.
- [LENV17] Massimo Lauria, Jan Elffers, Jakob Nordström, and Marc Vinyals. CNFgen: A generator of crafted benchmarks. In *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT '17)*, volume 10491 of *Lecture Notes in Computer Science*, pages 464–473. Springer, August 2017.
- [Lev73] Leonid A. Levin. Universal sequential search problems. *Problemy peredachi informatsii*, 9(3):115–116, 1973. In Russian. Available at <http://mi.mathnet.ru/ppi914>.
- [LGPC16] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for SAT solvers. In *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing (SAT '16)*, volume 9710 of *Lecture Notes in Computer Science*, pages 123–140. Springer, July 2016.
- [Lin] Lingeling, Plingeling and Treengeling. <http://fmv.jku.at/lingeling/>.
- [LMMW20] Daniel Le Berre, Pierre Marquis, Stefan Mengel, and Romain Walton. On irrelevant literals in pseudo-Boolean constraint learning. In *Proceedings of the 29th International Joint Conference on Artificial*

- Intelligence (IJCAI '20)*, pages 1148–1154, July 2020.
- [LMS02] Inês Lynce and João P. Marques-Silva. Building state-of-the-art SAT solvers. In *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI '02)*, pages 166–170, May 2002.
- [LMW20] Daniel Le Berre, Pierre Marquis, and Romain Wallon. On weakening strategies for PB solvers. In *Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing (SAT '20)*, volume 12178 of *Lecture Notes in Computer Science*, pages 322–331. Springer, July 2020.
- [LN17] Massimo Lauria and Jakob Nordström. Graph colouring is hard for algorithms based on Hilbert’s Nullstellensatz and Gröbner bases. In *Proceedings of the 32nd Annual Computational Complexity Conference (CCC '17)*, volume 79 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:20, July 2017.
- [LNSS20] Guillaume Lagarde, Jakob Nordström, Dmitry Sokolov, and Joseph Swernofsky. Trade-offs between size and degree in polynomial calculus. In *Proceedings of the 11th Innovations in Theoretical Computer Science Conference (ITCS '20)*, volume 151 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 72:1–72:16, January 2020.
- [LP10] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, July 2010.
- [LR18] Daniel Le Berre and Pascal Rapicault. Boolean-based dependency management for the Eclipse ecosystem. *International Journal on Artificial Intelligence Tools*, 27(1):1840003:1–1840003:23, February 2018.
- [LS91] László Lovász and Alexander Schrijver. Cones of matrices and set-functions and 0-1 optimization. *SIAM Journal on Optimization*, 1(2):166–190, 1991.
- [LT82] Thomas Lengauer and Robert Endre Tarjan. Asymptotically tight bounds on time-space trade-offs in a pebble game. *Journal of the ACM*, 29(4):1087–1130, October 1982. Preliminary version in *STOC '79*.
- [LTW18] Fu Li, Iddo Tzameret, and Zhengyu Wang. Characterizing propositional proofs as noncommutative formulas. *SIAM Journal on Computing*, 47(4):1424–1462, 2018.
- [Mar06] Klas Markström. Locality and hard SAT-instances. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):221–227, 2006.
- [MBCK18] Hakan Metin, Souheib Baarir, Maximilien Colange, and Fabrice Kordon. CDCLSym: Introducing effective symmetry breaking in SAT solving. In *Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 18)*, volume 10806 of *Lecture Notes in Computer Science*, pages 99–114. Springer, April 2018.
- [MBK19] Hakan Metin, Souheib Baarir, and Fabrice Kordon. Com-

- posing symmetry propagation and effective symmetry breaking for SAT solving. In *11th NASA Formal Methods Symposium (NFM '19)*, volume 11460 of *Lecture Notes in Computer Science*, pages 316–332. Springer, May 2019.
- [MHB13] Norbert Manthey, Marijn J. H. Heule, and Armin Biere. Automated reencoding of Boolean formulas. In *8th International Haifa Verification Conference (HVC '12), Revised Selected Papers*, volume 7857 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2013.
- [MIB⁺19] António Morgado, Alexey Ignatiev, Maria Luisa Bonet, João Marques-Silva, and Samula R. Buss. DRMaxSAT with MaxHS: First contact. In *Proceedings of the 22nd International Conference on Theory and Applications of Satisfiability Testing (SAT '19)*, volume 11628 of *Lecture Notes in Computer Science*, pages 239–249. Springer, July 2019.
- [MML14] Ruben Martins, Vasco M. Manquinho, and Inês Lynce. OpenWBO: A modular MaxSAT solver. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, volume 8561 of *Lecture Notes in Computer Science*, pages 438–445. Springer, July 2014.
- [MMS06] Vasco M. Manquinho and João Marques-Silva. On using cutting planes in pseudo-Boolean optimization. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:209–219, March 2006. Preliminary version in *SAT '05*.
- [MMZ⁺01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC '01)*, pages 530–535, June 2001.
- [MN14] Mladen Mikša and Jakob Nordström. Long proofs of (seemingly) simple formulas. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, volume 8561 of *Lecture Notes in Computer Science*, pages 121–137. Springer, July 2014.
- [MN15] Mladen Mikša and Jakob Nordström. A generalized method for proving polynomial calculus degree lower bounds. In *Proceedings of the 30th Annual Computational Complexity Conference (CCC '15)*, volume 33 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 467–487, June 2015.
- [MPR20] Nathan Mull, Shuo Pang, and Alexander A. Razborov. On CDCL-based proof systems with the ordered decision strategy. In *Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing (SAT '20)*, volume 12178 of *Lecture Notes in Computer Science*, pages 149–165. Springer, July 2020.
- [MPW02] Alexis Maciel, Toniann Pitassi, and Alan R. Woods. A new proof of the weak pigeonhole principle. *Journal of Computer and System Sciences*, 64(4):843–872, 2002. Preliminary version in *STOC '00*.
- [MPW19] Ian Mertz, Toniann Pitassi, and Yuanhao Wei. Short proofs are

- hard to find. In *Proceedings of the 46th International Colloquium on Automata, Languages and Programming (ICALP '19)*, volume 132 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 84:1–84:16, July 2019.
- [MSS99] João P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999. Preliminary version in *ICCAD '96*.
- [Nep70] Valery A. Nepomnjaščii. Rudimentary predicates and Turing computations. *Dokl. Akad. Nauk SSSR*, 195:282–284, 1970. English translation in *Soviet Math. Dokl.* 11 (1970) 1462–1465.
- [Nes00] Yurii Nesterov. Squared functional systems and optimization problems. In H. Frenk, K. Roos, T. Terlaky, and S. Zhang, editors, *High Performance Optimization*, pages 405–440. Kluwer Academic Publisher, 2000.
- [NH13] Jakob Nordström and Johan Håstad. Towards an optimal separation of space and length in resolution. *Theory of Computing*, 9:471–557, May 2013. Preliminary version in *STOC '08*.
- [Nor09a] Jakob Nordström. Narrow proofs may be spacious: Separating space and width in resolution. *SIAM Journal on Computing*, 39(1):59–121, May 2009. Preliminary version in *STOC '06*.
- [Nor09b] Jakob Nordström. A simplified way of proving trade-off results for resolution. *Information Processing Letters*, 109(18):1030–1035, August 2009.
- [Nor12] Jakob Nordström. On the relative strength of pebbling and resolution. *ACM Transactions on Computational Logic*, 13(2):16:1–16:43, April 2012. Preliminary version in *CCC '10*.
- [Nor13] Jakob Nordström. Pebble games, proof complexity and time-space trade-offs. *Logical Methods in Computer Science*, 9(3):15:1–15:63, September 2013.
- [Nor15] Jakob Nordström. On the interplay between proof complexity and SAT solving. *ACM SIGLOG News*, 2(3):19–44, July 2015.
- [Nor20] Jakob Nordström. New wine into old wineskins: A survey of some pebbling classics with supplemental results. Manuscript in preparation. To appear in *Foundations and Trends in Theoretical Computer Science*. Current draft version available at <http://www.csc.kth.se/~jakobn/research/>, 2020.
- [NR18] Alexander Nadel and Vadim Ryvchin. Chronological backtracking. In *Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing (SAT '18)*, volume 10929 of *Lecture Notes in Computer Science*, pages 111–121. Springer, July 2018.
- [Pan19] Shuo Pang. Large clique is hard on average for resolution. Technical Report TR19-068, Electronic Colloquium on Computational Complexity (ECCC), April 2019.
- [Par00] Pablo A. Parrilo. *Structured Semidefinite Programs and Semialgebraic Geometry Methods in Robustness and Optimiza-*

- tion. PhD thesis, California Institute of Technology, May 2000. Available at <http://resolver.caltech.edu/CaltechETD:etd-05062004-055516>.
- [PBI93] Toniann Pitassi, Paul Beame, and Russell Impagliazzo. Exponential lower bounds for the pigeonhole principle. *Computational Complexity*, 3:97–140, 1993. Preliminary version in *STOC '92*.
- [PD07] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Theory and Applications of Satisfiability Testing (SAT 2007)*, Lecture Notes in Computer Science 4501, pages 294–299. Springer Verlag, 2007.
- [PD11] Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning SAT solvers as resolution engines. *Artificial Intelligence*, 175(2):512–525, February 2011. Preliminary version in *CP '09*.
- [PF79] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *Journal of the ACM*, 26:361–381, 1979.
- [Pip80] Nicholas Pippenger. Pebbling. Technical Report RC8258, IBM Watson Research Center, 1980. In *Proceedings of the 5th IBM Symposium on Mathematical Foundations of Computer Science*.
- [PR17] Toniann Pitassi and Robert Robere. Strongly exponential lower bounds for monotone computation. In *Proceedings of the 49th Annual ACM Symposium on Theory of Computing (STOC '17)*, pages 1246–1255, June 2017.
- [PR18] Toniann Pitassi and Robert Robere. Lifting Nullstellensatz to monotone span programs over any field. In *Proceedings of the 50th Annual ACM Symposium on Theory of Computing (STOC '18)*, pages 1207–1219, June 2018.
- [PRST16] Toniann Pitassi, Benjamin Rossman, Rocco Servedio, and Li-Yang Tan. Poly-logarithmic Frege depth lower bounds. In *Proceedings of the 48th Annual ACM Symposium on Theory of Computing (STOC '16)*, pages 644–657, June 2016.
- [Pse16] Pseudo-Boolean competition 2016. <http://www.cril.univ-artois.fr/PB16/>, July 2016.
- [PT16] Toniann Pitassi and Iddo Zameret. Algebraic proof complexity: Progress, frontiers and challenges. *ACM SIGLOG News*, 3(3):21–43, August 2016.
- [PTC77] Wolfgang J. Paul, Robert Endre Tarjan, and James R. Celoni. Space bounds for a game on graphs. *Mathematical Systems Theory*, 10:239–251, 1977.
- [Pud97] Pavel Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *Journal of Symbolic Logic*, 62(3):981–998, September 1997.
- [Pud99] Pavel Pudlák. On the complexity of propositional calculus. In S. Barry Cooper and John K. Truss, editors, *Sets and Proofs*, volume 258 of *London Mathematical Society Lecture Note Series*, pages 197–218. Cambridge University Press, 1999.
- [PV05] Guoqiang Pan and Moshe Y. Vardi. Symbolic techniques in satisfiability solving. *Journal of Automated Reasoning*, 35(1–3):25–50, October 2005.

- [PW85] Jeff B. Paris and Alex J. Wilkie. Counting problems in bounded arithmetic. In *Methods in Mathematical Logic, Lecture Notes in Mathematics #1130*, pages 317–340. Springer-Verlag, 1985.
- [PWW88] Jeff B. Paris, Alex J. Wilkie, and A. R. Woods. Provability of the pigeonhole principle and the existence of infinitely many primes. *Journal of Symbolic Logic*, 53:1235–1244, 1988.
- [Raz87] Alexander A. Razborov. Lower bounds on the size of bounded depth networks over a complete basis with logical addition. *Matematicheskie Zametki*, 41(4):598–607, 1987. English Translation in *Mathematical Notes of the Academy of Sciences of the USSR*, 41(4):333338, 1987.
- [Raz98] Alexander A. Razborov. Lower bounds for the polynomial calculus. *Computational Complexity*, 7(4):291–324, December 1998.
- [Raz02] Alexander A. Razborov. Proof complexity of pigeonhole principles. In *5th International Conference on Developments in Language Theory, (DLT '01), Revised Papers*, volume 2295 of *Lecture Notes in Computer Science*, pages 100–116. Springer, July 2002.
- [Raz03] Alexander A. Razborov. Resolution lower bounds for the weak functional pigeonhole principle. *Theoretical Computer Science*, 1(303):233–243, June 2003.
- [Raz04a] Ran Raz. Resolution lower bounds for the weak pigeonhole principle. *Journal of the ACM*, 51(2):115–138, March 2004. Preliminary version in *STOC '02*.
- [Raz04b] Alexander A. Razborov. Resolution lower bounds for perfect matching principles. *Journal of Computer and System Sciences*, 69(1):3–27, August 2004. Preliminary version in *CCC '02*.
- [Raz15] Alexander A. Razborov. Pseudorandom generators hard for k -DNF resolution and polynomial calculus resolution. *Annals of Mathematics*, 181(2):415–472, March 2015.
- [Raz16a] Alexander A. Razborov. A new kind of tradeoffs in propositional proof complexity. *Journal of the ACM*, 63(2):16:1–16:14, April 2016.
- [Raz16b] Alexander A. Razborov. Proof complexity and beyond. *ACM SIGACT News*, 47(2):66–86, June 2016.
- [RBK17] Daniela Ritirc, Armin Biere, and Manuel Kauers. Column-wise verification of multipliers using computer algebra. In *Proceedings of the 17th International Conference on Formal Methods in Computer-Aided Design (FMCAD '17)*, pages 23–30, October 2017.
- [RBK18] Daniela Ritirc, Armin Biere, and Manuel Kauers. Improving and extending the algebraic approach for verifying gate-level multipliers. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE '18)*, pages 1556–1561, March 2018.
- [Rec75] Robert A. Reckhow. *On the Lengths of Proofs in the Propositional Calculus*. PhD thesis, University of Toronto, 1975. Available at https://www.cs.toronto.edu/~sacook/homepage/reckhow_thesis.pdf.

- [Rii93] Søren Riis. *Independence in Bounded Arithmetic*. PhD thesis, University of Oxford, 1993.
- [Rii97a] Soren Riis. Count(q) does not imply count(q). *Annals of Pure and Applied Logic*, 90:1–56, 1997.
- [Rii97b] Soren Riis. Count(q) versus the pigeon-hole principle. *Archive for Mathematical Logic*, 36:157–188, 1997.
- [RM99] Ran Raz and Pierre McKenzie. Separation of the monotone NC hierarchy. *Combinatorica*, 19(3):403–435, March 1999. Preliminary version in *FOCS '97*.
- [Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.
- [Rob79] John Michael Robson. A new proof of the NP-completeness of satisfiability. In *Proceedings of the 2nd Australian Computer Science Conference*, pages 62–70, February 1979.
- [Rob91] John Michael Robson. An $O(T \log T)$ reduction from RAM computations to satisfiability. *Theoretical Computer Science*, 82(1):141–149, May 1991.
- [RPRC16] Robert Robere, Toniann Pitassi, Benjamin Rossman, and Stephen A. Cook. Exponential lower bounds for monotone span programs. In *Proceedings of the 57th Annual IEEE Symposium on Foundations of Computer Science (FOCS '16)*, pages 406–415, October 2016.
- [RY20] Anup Rao and Amir Yehudayoff. *Communication Complexity and Applications*. Cambridge University Press, January 2020.
- [Rya04] Lawrence Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, February 2004. Available at <https://www.cs.sfu.ca/~mitchell/papers/ryan-thesis.ps>.
- [SA90] Hanif D. Sherali and Warren P. Adams. A hierarchy of relaxations between the continuous and convex hull representations for zero-one programming problems. *SIAM Journal on Discrete Mathematics*, 3:411–430, 1990.
- [SAT] The international SAT Competitions web page. <http://www.satcompetition.org>.
- [Sav98] John E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley, 1998. Available at <http://www.modelsofcomputation.org>.
- [SB06] Carsten Sinz and Armin Biere. Extended resolution proofs for conjoining BDDs. In *Proceedings of the 1st International Computer Science Symposium in Russia (CSR '06)*, volume 3967 of *Lecture Notes in Computer Science*, pages 600–611. Springer, June 2006.
- [SB09] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In *Theory and Applications of Satisfiability Testing (SAT 2009)*, Lecture Notes in Computer Science 5584, pages 237–243. Springer Verlag, 2009.
- [SBI04] Nathan Segerlind, Samula R. Buss, and Russell Impagliazzo. A switching lemma for small restrictions and lower bounds for k -DNF resolution. *SIAM Journal on Computing*, 33(5):1171–1200, 2004.

- Preliminary version in *FOCS '02*.
- [Sch78] Claus-Peter Schnorr. Satisfiability is quasilinear complete in NQL. *Journal of the ACM*, 25(1):136–145, January 1978.
- [SCI] SCIP: Solving constraint integer programs. <http://scip.zib.de/>.
- [SDNS20] Buser Say, Jo Devriendt, Jakob Nordström, and Peter Stuckey. Theoretical and experimental results for planning with learned binarized neural network transition models. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP '20)*, volume 12333 of *Lecture Notes in Computer Science*, pages 917–934. Springer, September 2020.
- [Seg07] Nathan Segerlind. The complexity of propositional proofs. *Bulletin of Symbolic Logic*, 13(4):417–481, December 2007.
- [Seg08] Nathan Segerlind. On the relative efficiency of resolution-like proofs and ordered binary decision diagram proofs. In *Proceedings of the 23rd Annual IEEE Conference on Computational Complexity (CCC '08)*, pages 100–111, June 2008.
- [Sho87] Naum Z. Shor. An approach to obtaining global extremums in polynomial mathematical programming problems. *Cybernetics*, 23(5):695–700, 1987. Translated from *Kibernetika*, No. 5, pages 102–106, 1987.
- [Sim14] Laurent Simon. Post mortem analysis of SAT solver proofs. In *Proceedings of the 5th Pragmatics of SAT workshop*, volume 27 of *EPiC Series in Computing*, pages 26–40, July 2014. Available at <https://easychair.org/publications/paper/N3GD>.
- [Smo87] Roman Smolensky. Algebraic methods in the theory of lower bounds for Boolean circuit complexity. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC '87)*, pages 77–82, 1987.
- [SN15] Masahiko Sakai and Hidetomo Nabeshima. Construction of an ROBDD for a PB-constraint in band form and related techniques for PB-solvers. *IEICE Transactions on Information and Systems*, 98-D(6):1121–1127, June 2015.
- [Spe10] Ivor Spence. sgen1: A generator of small but difficult satisfiability benchmarks. *Journal of Experimental Algorithmics*, 15:1. 2:1–1. 2:15, March 2010.
- [SS06] Hossein M. Sheini and Karem A. Sakallah. Pueblo: A hybrid pseudo-Boolean SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):165–189, March 2006. Preliminary version in *DATE '05*.
- [Stå96] Gunnar Stålmarmark. Short resolution proofs for a sequence of tricky formulas. *Acta Informatica*, 33(3):277–280, May 1996.
- [Sze05] Stefan Szeider. The complexity of resolution with generalized symmetry rules. *Theory of Computing Systems*, 38(2):171–188, January 2005. Preliminary version in *STACS '03*.
- [TC17] Iddo Zameret and Stephen A. Cook. Uniform, integral and efficient proofs for the determinant identities. In *Proceedings of the 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '17)*, pages 1–12, June 2017.

- [TD19] Rodrigue Konan Tchinda and Clémentin Tayou Djamégni. Enhancing static symmetry breaking with dynamic symmetry handling in CDCL SAT solvers. *International Journal on Artificial Intelligence Tools*, 28(3):1950011:1–1950011:32, May 2019.
- [Tha16] Neil Thapen. A trade-off between length and width in resolution. *Theory of Computing*, 12(5):1–14, August 2016.
- [Tse68] Grigori Tseitin. The complexity of a deduction in the propositional predicate calculus. *Zapiski Nauchnyh Seminarov Leningradskogo Otdelenija matematicheskogo Instituta im. V. A. Steklova akademii Nauk SSSR (LOMI)*, 8:234–259, 1968. In Russian.
- [TSZ10] Olga Tveretina, Carsten Sinz, and Hans Zantema. Ordered binary decision diagrams, pigeonhole formulas and beyond. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(1):35–58, March 2010.
- [Urq87] Alasdair Urquhart. Hard examples for resolution. *Journal of the ACM*, 34(1):209–219, January 1987.
- [Urq99] Alasdair Urquhart. The symmetry rule in propositional logic. *Discrete Applied Mathematics*, 96–97:177–193, October 1999.
- [Urq11] Alasdair Urquhart. A near-optimal separation of regular and general resolution. *SIAM Journal on Computing*, 40(1):107–121, 2011. Preliminary version in *SAT '08*.
- [Van05] Allen Van Gelder. Pool resolution and its relation to regular resolution and DPLL with clause learning. In *Proceedings of the 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR '05)*, volume 3835 of *Lecture Notes in Computer Science*, pages 580–594. Springer, 2005.
- [Van08] Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *10th International Symposium on Artificial Intelligence and Mathematics (ISAIM '08)*, 2008. Available at <http://isaim2008.unl.edu/index.php?page=proceedings>.
- [VEGC⁺18] Marc Vinyals, Jan Elffers, Jesús Giráldez-Cru, Stephan Gocht, and Jakob Nordström. In between resolution and cutting planes: A study of proof systems for pseudo-Boolean SAT solving. In *Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing (SAT '18)*, volume 10929 of *Lecture Notes in Computer Science*, pages 292–310. Springer, July 2018.
- [Ver19] VeriPB: Verifier for pseudo-Boolean proofs. <https://doi.org/10.5281/zenodo.3548581>, 2019.
- [Vin20] Marc Vinyals. Hard examples for common variable decision heuristics. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI '20)*, pages 1652–1659, February 2020.
- [VS10] Allen Van Gelder and Ivor Spence. Zero-one designs produce small hard SAT instances. In *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing (SAT '10)*, volume 6175 of *Lecture Notes in Computer Science*, pages 388–397. Springer, July 2010.

Chapter 8

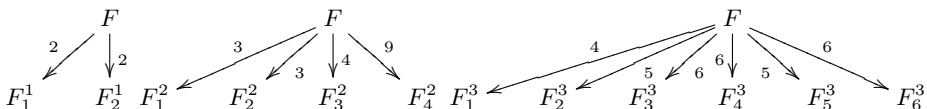
Fundamentals of Branching Heuristics

Oliver Kullmann

8.1. Introduction

The topic of this chapter is to provide foundations for “branching heuristics”. The whole field of “heuristics” is very diverse, and we will concentrate on a specific part, where developments in the last four decades can be comprised in what actually deserves to be called a “theory”. A full version of this chapter, containing all proofs and extensive examples, is available in [Kul08a].

The notion of a “heuristics” is fundamental for the field of Artificial Intelligence. The (early) history of the notion of “heuristics” is discussed in [BF81], Section II.A, and the usage of this notion in the SAT literature follows their definition of a heuristics as a method using additional information to restrict the search space size, though in this chapter we consider a restricted context, where completeness is not an issue, but the heuristical component of the search process affects only resource usage, not correctness. Furthermore we only study a specific form of search processes here, namely backtracking search. We consider the situation where we have a problem instance F where all direct (“efficient”) methods fail, and so F has to be split into subproblems. In the context of this chapter we basically assume that the method for splitting F is already given, yielding possible “branchings” $F \rightsquigarrow F_1, \dots, F_m$, splitting F into m “subproblems” F_i , and the task of the heuristic is to compare different branchings and to find the “best” branching among them. Let us assume that we have given three branchings to compare:



We see an important aspect of our abstract analysis: Each branch is labelled by a positive real number, which measures the “distance”, that is, how much “simpler” the problem became (w.r.t. a certain aspect). At this level of abstraction not only the branchings but also these distances are given, and the question is what can be done with these numbers, how can they be used to compare these three

branchings? Obviously, all that counts then are the tuples of distances, so-called “branching tuples”. We get $a := (2, 2)$, $b := (3, 3, 4, 9)$, $c := (4, 5, 6, 6, 5, 6)$.

The first part of this chapter is devoted to the combinatorics and analysis of branching tuples. We will see that a canonical value $\tau(t) \in \mathbb{R}_{\geq 1}$ (see Definition 8.3.2) can be computed for each branching tuple, and that these values under fairly general circumstances yield the (only) answer to the problem of comparing branching tuples: The smaller the τ -value the better the tuple. For the above branching tuples the computation yields $\tau(a) = 1.4142\dots$, $\tau(b) = 1.4147\dots$, and $\tau(c) = 1.4082\dots$, and thus the third branching is the best, followed by the first branching, and then the second branching (all under the assumption that all that is given are these numbers). The τ -function arises naturally from standard techniques from difference equations, and for example $\tau(b)$ is the (unique) positive solution of $x^{-3} + x^{-3} + x^{-4} + x^{-9} = 1$. Using an appropriate scaling, the τ -function yields a (generalised) mean \mathfrak{T} (in a precise sense), and the task of evaluating branchings can be understood as extracting a generalised mean-value from each tuple (where in this interpretation the tuple is the better the larger the mean). It is worth mentioning here that branching tuples are arbitrary tuples of positive *real* numbers, and thus are amenable to optimisation techniques.

After having developed a reasonable understanding of branching tuples, based on the analysis of *rooted trees* which consist of these branching tuples w.r.t. a run of the given backtracking solver, we then turn to the question of how actually to compute “good” distances. At this time the general theory can only give rough guidelines, which however suffices to *compare* different distance functions w.r.t. their appropriateness. So we can compare distance functions, and we can also optimise them. But finally we have to come up with some concrete distance, and fortunately, in the second part of this chapter, at least for CNF in the context of so-called “look-ahead” solvers a reasonable answer can be given, obtained by a remarkable convergence of theoretical and empirical studies. We need to remark here that heuristics for so-called “conflict-driven” solvers cannot be fully handled here, basically since these solvers are not really based on backtracking anymore, that is, we cannot speak of *independent* branches anymore, but rather an iterative process is taking place.

In more details, the content of this chapter is as follows:

1. The general framework and its assumptions are discussed in Section 8.2. First we discuss the focus of our approach, and which questions are excluded by this theory, which is based on comparing branchings by condensing them into branching tuples. The way these tuples arise is based on “measures” (of approximated problem complexity) or more generally on “distances”, and we discuss the meaning of this process of extracting branching tuples. Alternative branchings are represented by alternative branching tuples, and so we need to order branching tuples by some linear quasi-order (smaller is better), choosing then a branching with smallest associated branching tuple. This order on the set of branching tuples is given by applying a “projection function”, and by an introductory example

- we motivate the “canonical projection”, the τ -function.
2. The combinatorics of branching tuples and the fundamental properties of the τ -function are the subject of Section 8.3. In particular we consider bounds on the τ -function (resp. on the associated mean \mathfrak{T}). These bounds are important to derive upper and lower bounds on tree sizes. And under special circumstances one might wish to consider alternatives to the τ -function as a “projection” (comprising a branching tuple to one number), and then these bounds provide first alternatives. Section 8.3 is concluded by introducing the fundamental association of probabilities with branching tuples.
 3. Then in Section 8.4 we discuss the fundamental method of estimating tree sizes, first for trees with given probability distributions, and then for trees where the probability distribution is derived from a given distance by means of the τ -function.
 4. The τ -function is “in general” the only way to comprise branching tuples into a single number, however the precise value of the τ -function is not of importance, only the linear quasi-order it induces on the set of all branching tuples; this is proven in Section 8.5.
 5. Though the τ -function is the canonical projection in general, there might be reasons to deviate from it under special circumstances. The known facts are presented in Section 8.6. For binary branching tuples (t_1, t_2) (dominant for practical SAT solving) we relate the projections $t_1 + t_2$ and $t_1 \cdot t_2$ to the τ -function, yielding a strong analytical argument why the “product rule” is better than the “sum rule”, complementing the experimental evidence.
 6. With Section 8.7 we enter the second part of this chapter, and we discuss the known distances w.r.t. practical SAT solving.
 7. As already mentioned, at present there is no theory for choosing a “good” distance for a particular class of problem instances (other than the trivial choice of the optimal distance), but from the general theory developed in the first part of this chapter we obtain methods for improving distance functions, and this is discussed in Section 8.8.
 8. Our approach on branching is based on a two-phase model, where first the branching itself is chosen, and then, in a second step, the order of the branches. Methods for finding good orders (for SAT problems) are discussed in Section 8.9.
 9. The ideas for concrete distances, as presented in Section 8.7, also have bearings on more general situations than just boolean CNF-SAT, and especially our general theory is applicable in a much larger context. A quick outlook on this topic is given in Section 8.10.

8.2. A general framework for branching algorithms

A general framework for heuristics for branching algorithms is as follows: Consider a non-deterministic machine \mathcal{M} for solving some problem, where the computation terminates and is correct on *each* possible branch, and thus the decisions made

during the run of the machine only influence resource consumption. The task for a heuristics \mathcal{H} is to make the machine deterministic, that is, at each choice point to choose one of the possible branches, obtaining a deterministic machine $\mathcal{M}_{\mathcal{H}}$, where typically time and/or space usage is to be minimised. Likely not much can be said about the choice-problem in this generality, since no information is given about the choices. The focus of this article is on the problem of good choices between different possibilities of splitting problems into (similar) *subproblems*, where for each possible choice (i.e., for each possible splitting) we have (reasonable) information about the subproblems created. Not all relevant information usable to gauge branching processes for SAT solving can be represented (well) in this way, for example non-local information is hard to integrate into this “recursive” picture, but we consider the splitting-information as the central piece, while other aspects are treated as “add-ons”.

8.2.1. Evaluating branchings

The basic scenario is that at the current node v of the backtracking tree we have a selection $\mathcal{B}(v) = (B_1, \dots, B_m)$ of branchings given, and the heuristic chooses one. Each branching is (in this abstract framework) considered as a tuple $B_i = (b_1, \dots, b_k)$ of branches, where each b_i is a “smaller” problem instance, and k is the width of the branching. If the determination of the order of branches is part of the heuristics, then all $k!$ permutations of a branching are included in the list $\mathcal{B}(v)$, otherwise a standard ordering of branches is chosen. If we consider branching on a boolean variable, where the problem instance contains n variables, and all of them are considered by the heuristics, then the selection $\mathcal{B}(v)$ contains $2n$ branchings if the order of the two branches is taken into account, and only n branchings otherwise.

The problem is solved in principle, if we have precise (or “good”) knowledge about the resource consumption of the subproblems b_i (in the order they are processed, where the running time of b_i might depend on the running time of b_j for $j < i$), since then for every possible branching we sum up the running times of the branches (which might be 0 if the branch is not executed) to get the total running time for this branching, and we choose a branching with minimal running time. If ordinary backtracking order is not followed (e.g., using restarts, or some form of evaluating the backtracking tree in some other order), or branches influence other branches (e.g., due to learning), then this might be included in this picture by the assumption of “complete clairvoyance”.

Though this picture is appealing, I am not aware of any circumstances in (general) SAT solving where we actually have good enough knowledge about the resource consumption of the subproblems b_i to apply this approach successfully. Even in the probabilistically well-defined and rather restricted setting of random 3-SAT problems, a considerable effort in [Ouy99] (Chapter 5) to construct such a “rational branching rule” did not yield the expected results. The first step towards a practical solution is to use (rough) estimates of problem complexity, captured by a measure $\mu(F)$ of “problem complexity”. We view $\mu(F)$ as a kind of

logarithm of the true complexity. For example, the trivial SAT algorithm has the bound $2^{n(F)}$, and taking the logarithm (base 2) we obtain the most basic measure of problem complexity here, the number $n(F)$ of variables. This “logarithmic” point of view is motivated by the optimality result Lemma 8.4.9. Progress in one branch $F \rightsquigarrow F'$ then can be measured by $\Delta\mu(F, F') = \mu(F) - \mu(F') > 0$. However, since at this time the practical measures $\mu(F)$ are too rough for good results, instead of the difference $\Delta\mu(F, F')$ a more general “distance” $d(F, F') > 0$ needs to be involved, which estimates, in some heuristic way, the prospects F' offers to actually be useful in the (near) future (relative to F , for the special methods of the algorithm considered). Before outlining the framework for this kind of analysis, two basic assumptions need to be discussed:

- A basic assumption about the estimation of branching quality is *homogeneity*: The branching situation might occur, appropriately “relocated”, at many other places in the search tree, and is not just a “one-off” situation. If we have a “special situation” at hand (and we are aware of it), then, in this theoretical framework, handling of this special situation is not the task of the heuristics, but of the “reduction” for the currently given problem (compare Subsection 8.7.2).
- Another abstraction is, that as for theoretical upper-bounds, a “mathematical running time” is considered: The essential abstraction is given by the *search tree*, where we ignore what happens “inside a node”, and then the mathematical running time is the number of nodes in this tree. Real running times (on real machines) are not considered by (current, abstract) heuristics. In the literature (for example in [Ouy99]) one finds the attempt of taking the different workloads at different nodes into account by measuring the (total) number of assignments to variables, but this works only for special situations, and cannot be used in general. Furthermore, practice shows that typically, given the number of nodes in the search tree and some basic parameters about the problem size, curve-fitting methods yield good results on predicting the actual running time of a solver.¹

8.2.2. Enumeration trees

We focus on algorithms where an enumeration tree is built, and where the main heuristical problem is how to control the growth of the tree.

1. We do not consider “restarts” here, that is, rebuilding the tree, possibly learning from prior experiences. In the previous phase of SAT-usage, emphasise was put on *random* restarts, with the aim of undoing bad choices, while in the current phase randomness seems no longer important, but the effects of *learning* are emphasised. In this sense, as remarked by John Franco, restarts can be seen as a kind of “look-ahead”.

¹The framework we develop here actually is able to handle different run times at different nodes by attaching different weights to nodes. However in this chapter, where the emphasise is on setting up the basic framework, we do not consider this; see [Kul08a] for how to generalise the framework.

2. We assume mostly a depth-first strategy.
3. “Intelligent backtracking” (that is, not investigating the second branch in the case of an unsatisfiable first branch where the conflict does not depend on the branching) is considered as accidental (not predictable), and thus not included in heuristical considerations.
4. “Jumping around” in the search tree in order to prioritise nodes with higher probability of finding a satisfying assignment (like in [Hv08]) is considered as a layer on top of the heuristics (a kind of control layer), and is not considered by the core heuristics (given the current state of research).
5. The effect one branch might have on others (via “global learning”) is also considered a secondary effect, out of our control and thus not directly covered by the heuristics.
6. As discussed in [Kul08b], conflict-driven solvers actually tend to choose an iterative approach, not creating independent sub-problems for branching, but just choosing one branch and leaving it to learning to cater for completeness. Such a process seems currently unanalysable.

After having outlined what is to be included in our analysis, and what is to be left out, we now give a first sketch of the framework to be developed in this article.

8.2.3. A theoretical framework

For the author of this chapter, the theory developed here originated in the theoretical considerations for proving upper bounds on SAT decision (see Section 8.7.3 for further comments), and so the underlying “driving force” of the heuristics is the minimisation of an upper bound on the search tree size. Only later did I see that in [Knu75] actually the same situation is considered, only from a different point of view, a probabilistic one where a path through the search tree is chosen by randomly selecting one successor at each inner node.²

Example 8.2.1. An instructive example is the first non-trivial 3-SAT bound ([Luc84, MS85]), where via the autarky argument it can be assumed that a binary clause is always present, and then splitting on a variable in this binary clauses eliminates one variable in one branch and two variables in the other (due to unit-clause elimination). “Traditionally” this is handled by considering the worst case and using a difference equation $f_n = f_{n-1} + f_{n-2}$ (n is the number of variables, while f_n is the number of leaves in the search tree). Reasonably we assume $f_0 = 0$ and $f_1 = 1$, and then $(f_n)_{n \in \mathbb{N}_0}$ is the Fibonacci sequence with $f_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right)$, and using $r : \mathbb{R} \setminus \left(\frac{1}{2} + \mathbb{Z} \right) \rightarrow \mathbb{Z}$ for rounding to the nearest integer, we have $f_n = r \left(\frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n \right)$ (see Chapter 1 in [CFR05] for an introduction into these considerations). Whatever the initial values f_0, f_1 are, we always have $f_n = \Theta \left(\left(\frac{1+\sqrt{5}}{2} \right)^n \right) = O(1.619^n)$. The fundamental approach is the Ansatz $f_n = \lambda^n$, which leads to the equation $\lambda^n = \lambda^{n-1} + \lambda^{n-2}$, which is equivalent to $\lambda^2 = \lambda + 1$.

²In fact, the proof of the central “ τ -lemma” in [Kul99b] used already the probabilistic interpretation, but [Knu75] was not known to me at this time.

We do not follow the general theory of difference equations any further, since the initial conditions are of no relevance, and we also want to allow descents in arbitrary positive *real* numbers, in order to allow optimisations. The computation of the root λ from Example 8.2.1 will be generalised in Definition 8.3.2, while Theorem 8.4.8 yields a general method for computing bounds on tree sizes. Regarding our main purpose, the evaluation of branchings, we can show (in Section 8.5) that this generalised λ -calculation is the only “general” way of projecting a branching tuple to a single number.

As we have already mentioned, the first step is to move from branchings to branching tuples. For Example 8.2.1 this means extracting the branching tuple $(1, 2)$ as the essential piece of information. Now an important aspect of the theory developed in this chapter is that branching tuples are not only considered in isolation, but in connection with the (search) trees (in abstracted form). This “emancipation” of search trees, which in the form of worst-case analysis based on recurrence equations are factually suppressed, is also important for practical applications, as demonstrated in Section 8.8, since it allows to consider the real computations going on, not just their shadows in the form of worst-case considerations, which must deliver some interpretable bound at the end. And furthermore the study of probability distributions on the leaves of the tree combine the worst-case upper bounds with the probabilistic considerations from [Knu75] — the λ -calculation associates, in a canonical way, a probability distribution to the branches of a branching tuple (see Subsection 8.3.4).

So a main part of the theory is concerned with estimating sizes of trees in connection with branching tuples associated with them, based on simple probability considerations. For this task, the above λ -calculation takes on the more general form of evaluating a branching tuple by a real number. We call such evaluations *evaluation projection*, but actually, since the only form of projections used in this paper are such evaluation projections, we just call them “projections”. The general study of projections yields the underlying theory to answer questions like “Why is the product-rule for SAT heuristics better than the sum-rule?” (see Section 8.6). We begin the development of the theoretical framework by taking a closer look at branching tuples and the “ λ -calculation”.

8.3. Branching tuples and the canonical projection

The subject of this section is the theory of “branching tuples” and their evaluation. The theory shows its full strength when considering branchings of arbitrary width, which at this time is of importance for theoretical upper bounds (see Section 8.7.3) and for constraint satisfaction (see Section 8.10.2), and which might become more important for practical SAT solving when for example considering deeper look-aheads at the root of the search tree.

8.3.1. Branching tuples and their operations

One single (potential) branching consisting of $k \in \mathbb{N}$ branches is evaluated by a “branching tuple” a of length k , attaching to each branch i a positive real num-

ber a_i , which is intended to measure the progress achieved in reducing problem complexity in this branch (thus the larger a_i , the better is branch i).

Definition 8.3.1. $\mathcal{BT} := \bigcup_{k \in \mathbb{N}} (\mathbb{R}_{>0})^k$ denotes the set of **branching tuples**.

Remarks:

1. Basic measurements for branching tuples are minimum $\min : \mathcal{BT} \rightarrow \mathbb{R}_{>0}$, maximum $\max : \mathcal{BT} \rightarrow \mathbb{R}_{>0}$, sum $\Sigma : \mathcal{BT} \rightarrow \mathbb{R}_{>0}$, and width $\| : \mathcal{BT} \rightarrow \mathbb{N}$. The minimum $\min(a)$ of a branching tuple is a “worst-case view”, while $\max(a)$ is a “best-case view”. In general, disregarding the values, the larger $|a|$, i.e., the wider the branching is, the worse it is.
2. The set of branching tuples of width k is $\mathcal{BT}^{(k)} := \{t \in \mathcal{BT} : |t| = k\}$, which is a *cone*, that is for $a \in \mathcal{BT}^{(k)}$ and $\lambda \in \mathbb{R}_{>0}$ we have $\lambda \cdot t \in \mathcal{BT}^{(k)}$, and for $a, b \in \mathcal{BT}^{(k)}$ we have $a + b \in \mathcal{BT}^{(k)}$.
3. Branching tuples of width 1, which do not represent “real branchings” but “reductions”, are convenient to allow.
4. One could also allow the empty branching tuple as well as the zero branching tuple (0) (of width 1), but for the sake of simplicity we abstain from such systematic extensions here.

Concatenation of branching tuples a, b is denoted by “ $a ; b$ ”, and yields the semigroup $(\mathcal{BT}, ;)$ (the empty branching tuple would be the neutral element here). The width function $\|$ now becomes a homomorphism from $(\mathcal{BT}, ;)$ to $(\mathbb{N}, +)$. Concatenation allows us to define the strict prefix order $a \sqsubset b \Leftrightarrow \exists x \in \mathcal{BT} : a ; x = b$ (that is, b is obtained from a by adding further positive numbers to the end of a) for $a, b \in \mathcal{BT}$, while $a \sqsubseteq b \Leftrightarrow a \sqsubset b \vee a = b$. A further basic operation for a branching tuple a of width k is to apply a *permutation* $\pi \in S_k$, which we denote by $\pi * a := (a_{\pi(1)}, \dots, a_{\pi(k)})$. Finally we have **composition of branching tuples** a, b at position i of a , that is, branching b is attached to branch i of a ; since we allow permutation of branching tuples, it suffices to set $i = 1$ here, and the resulting composition is denoted by $\mathbf{a} \mathbb{M} \mathbf{b}$, defined as

$$(a_1, \dots, a_p) \mathbb{M} (b_1, \dots, b_q) := (a_1 + b_1, \dots, a_1 + b_q, a_2, \dots, a_p)$$

$$=$$

We obtain a semigroup $(\mathcal{BT}, \mathbb{M})$ (non-commutative; the zero branching tuple (0) would be the neutral element). It is important to realise that it makes a difference where to attach the branching b , that is in our setting, the branching tuples $a \mathbb{M} b$ and $(\pi * a) \mathbb{M} b$ are in general qualitatively different:

- (i) if b is better than a , then attaching b to a smaller component of a yields a better tuple than when attaching it to a larger component;
- (ii) if b is worse than a , then attaching b to a smaller component of a yields a worse tuple than when attaching it to a larger component.

The intuitive reason is that “more balanced tuples are better”, and so in the first case it is a greater improvement to a when improving its weak parts than when improving its strong parts, while in the second case making a weak part worse means a greater impairment than making a strong part worse.

8.3.2. The tau-function

In this section we introduce formally the τ -function as discussed in Example 8.2.1, and show its main properties.

Definition 8.3.2. Define $\chi_k : \mathcal{BT} \times \mathbb{R}_{>0} \rightarrow \mathbb{R}_{>0}$ by $\chi(t, x) := \sum_{i=1}^{|t|} x^{-t_i}$. Observe that for each $t \in \mathcal{BT}$ the map $\chi(t, -) : \mathbb{R}_{>0} \rightarrow \mathbb{R}_{>0}$ is strictly decreasing with $\chi(t, 1) = |t| \geq 1$ and $\lim_{x \rightarrow \infty} \chi(t, x) = 0$. Now $\tau : \mathcal{BT} \rightarrow \mathbb{R}_{\geq 1}$ is defined as the unique $\tau(t) := x_0 \in \mathbb{R}_{\geq 1}$ such that $\chi(t)(x_0) = 1$ holds.

By definition we have $\tau(t) \geq 1$, with $\tau(t) = 1 \Leftrightarrow |t| = 1$. For $k \in \mathbb{N}$ we denote by $\tau_k : \mathcal{BT}^{(k)} \rightarrow \mathbb{R}_{\geq 1}$ the τ -function restricted to branching tuples of width k .

Example 8.3.1. The computation of $\tau(1, 2)$ (recall Example 8.2.1) leads to the equation $x^{-1} + x^{-2} = 1$, which is equivalent to $x^2 - x - 1 = 0$, which has the two solutions $\frac{1 \pm \sqrt{5}}{2}$, and thus $\tau(1, 2) = \frac{1 + \sqrt{5}}{2} = 1.6180\dots$ Only very few τ -values can be expressed by analytical formulas, and most of the time numerical computations have to be used (see Remark 2 to Corollary 8.3.5), so for example $\tau(2, 3, 5) = 1.4291\dots$

Simple properties can be proven directly:

Lemma 8.3.1. *For every $a \in \mathcal{BT}$, $k \in \mathbb{N}$ and $\lambda \in \mathbb{R}_{>0}$ we have:*

1. $\tau(\lambda \cdot a) = \tau(a)^{1/\lambda}$.
2. $\tau_k(\vec{1}) = k$.
3. τ_k for $k \geq 2$ is strictly decreasing in each component.
4. τ_k is symmetric, that is, invariant under permutation of branching tuples.
5. $\tau(a)^{\min(a)} \leq |a| \leq \tau(a)^{\max(a)}$, that is, $|a|^{1/\max(a)} \leq \tau(a) \leq |a|^{1/\min(a)}$.
6. $\lim_{\lambda \rightarrow 0} \tau(a; (\lambda)) = \infty$ and $\lim_{\lambda \rightarrow \infty} \tau(a; (\lambda)) = \tau(a)$.

The τ -function fulfils powerful convexity properties, from which non-trivial further properties will follow. A function $f : C \rightarrow \mathbb{R}$ defined on some convex subset $C \subseteq \mathbb{R}^k$ is called “strictly convex” if for all $x, y \in C$ and $0 < \lambda < 1$ holds $f(\lambda x + (1 - \lambda)y) < \lambda f(x) + (1 - \lambda)f(y)$; furthermore f is called “strictly concave” if $-f$ is strictly convex. By definition τ_1 is just the constant function with value 1, and so doesn’t need to be considered here.

Lemma 8.3.2. *For $k \geq 2$ the function τ_k is strictly convex.*

Lemma 8.3.2 strengthens considerably the quasi-convexity of τ_k as shown in Lemma 4.1 of [Epp06]; in Corollary 8.3.5 we shall prove a further strengthening.

8.3.3. Bounds on the tau-function

Just from being symmetric and strictly convex it follows, that $\tau_k(a)$ for tuples a with a given fixed sum $\Sigma(a) = s$ attains its strict minimum for the constant tuple (with entries $\frac{s}{k}$); see Lemma 8.3.3 below for a proof. Thus, using $\mathfrak{A}(t) := \Sigma(t)/|t|$ for the arithmetic mean of a branching tuple, we have $\tau(\mathfrak{A}(t) \cdot \vec{1}) \leq \tau(t)$ (with strict inequality iff t is not constant). In the remainder of this subsection we will be concerned with further estimations on the τ -functions, and for that purpose we use the following well-known means (see [HLP99, Bul03]):

1. The *arithmetic mean*, the *geometric mean*, and the *harmonic mean* of branching tuples t are denoted respectively by $\mathfrak{A}(t) := \frac{1}{|t|}\Sigma(t) = \frac{1}{|t|} \sum_{i=1}^{|t|} t_i$, $\mathfrak{G}(t) := \sqrt[|t|]{\prod_{i=1}^{|t|} t_i}$ and $\mathfrak{H}(t) := |t| / (\sum_{i=1}^{|t|} \frac{1}{t_i})$. We recall the well-known fundamental inequality between these means: $\mathfrak{H}(t) \leq \mathfrak{G}(t) \leq \mathfrak{A}(t)$ (where strict inequalities hold iff t is not constant).
2. More generally we have the *power means* for $\alpha \in \overline{\mathbb{R}}$ given by $\mathfrak{M}_\alpha(t) := (\frac{1}{|t|} \sum_{i=1}^{|t|} t_i^\alpha)^{1/\alpha}$ for $\alpha \notin \{-\infty, 0, +\infty\}$, while we set $\mathfrak{M}_{-\infty}(t) := \min(t)$, $\mathfrak{M}_0(t) := \mathfrak{G}(t)$ and $\mathfrak{M}_{+\infty}(t) := \max(t)$. By definition we have $\mathfrak{M}_{-1}(t) = \mathfrak{H}(t)$ and $\mathfrak{M}_1(t) = \mathfrak{A}(t)$. In generalisation of the above fundamental inequality we have for $\alpha, \alpha' \in \overline{\mathbb{R}}$ with $\alpha < \alpha'$ the inequality $\mathfrak{M}_\alpha(t) \leq \mathfrak{M}_{\alpha'}(t)$, which is strict iff t is not constant.

We want to establish a variation of the τ -function as a “mean”, comparable to the above means. In the literature there are no standard axiomatic notions about “means”, only collections of relevant properties, and we condense the relevant notion here as follows:

Definition 8.3.3. Consider $k \in \mathbb{N}$. A **mean** is a map $M : \mathcal{BT}^{(k)} \rightarrow \mathbb{R}_{>0}$ which is continuous, strictly monotonic increasing in each coordinate, symmetric (i.e., invariant under permutation of the arguments) and “consistent”, that is, $\min(a) \leq M(a) \leq \max(a)$ for $a \in \mathcal{BT}^{(k)}$. A mean M is **homogeneous** if M is positive homogeneous, i.e., for $\lambda \in \mathbb{R}_{>0}$ and $a \in \mathcal{BT}^{(k)}$ we have $M(\lambda \cdot a) = \lambda \cdot M(a)$.

All power means are homogeneous means. Yet k -ary means M are only defined for tuples of positive real numbers $a \in \mathbb{R}_{>0}^k$, and we extend this as follows to allow arguments 0 or $+\infty$, using the extended real line $\overline{\mathbb{R}} = \mathbb{R} \cup \{\pm\infty\}$. We say that M is defined for $a \in \overline{\mathbb{R}}_{\geq 0}^k$ (allowing positions to be 0 or $+\infty$) if the limit $\lim_{a' \rightarrow a, a' \in \mathbb{R}_{>0}^k} M(a')$ exists in $\overline{\mathbb{R}}$, and we denote this limit by $M(a)$ (if the limit exists, then it is unique). Power means $\mathfrak{M}_\lambda(a)$ for $\lambda \neq 0$ are defined for all $a \in \overline{\mathbb{R}}_{\geq 0}^k$, while $\mathfrak{M}_0(a) = \mathfrak{G}(a)$ is defined iff there are no indices i, j with $a_i = 0$ and $a_j = \infty$.

Definition 8.3.4. Consider a mean $M : \mathcal{BT}^{(k)} \rightarrow \mathbb{R}_{>0}$. We say that M is **∞ -dominated** resp. **0-dominated** if for every $a \in \overline{\mathbb{R}}_{\geq 0}^k$, such that an index i with $a_i = \infty$ resp. $a_i = 0$ exists, $M(a)$ is defined with $M(a) = \infty$ resp. $M(a) = 0$. On the other hand, M **ignores ∞** resp. **ignores 0** if $M(a; (\infty))$ resp. $M(a; (0))$ is defined iff $M(a)$ is defined with $M(a; (\infty)) = M(a)$ resp. $M(a; (0)) = M(a)$.

Power means \mathfrak{M}_λ with $\lambda > 0$ are ∞ -dominated and ignore 0, while for $\lambda < 0$ they are 0-dominated and ignore ∞ . The borderline case $\mathfrak{M}_0 = \mathfrak{G}$ is ∞ -dominated as well as 0-dominated if only tuples are considered for which \mathfrak{G} is defined (and thus we do not have to evaluate “ $0 \cdot \infty$ ”).

Another important properties of means is convexity resp. concavity. Power means \mathfrak{M}_α with $\alpha > 1$ are strictly convex, while power means \mathfrak{M}_α with $\alpha < 1$ are strictly concave; the borderline case $\mathfrak{M}_1 = \mathfrak{A}$ is linear (convex and concave).

Lemma 8.3.3. *For every concave mean M we have $M \leq \mathfrak{A}$.*

Proof. By Jensen’s inequality (see [HUL04]) we have $M(a) = \sum_{\pi \in S_k} \frac{1}{k!} M(\pi * a) \leq M(\sum_{\pi \in S_k} \frac{1}{k!} \cdot (\pi * a)) = M(\mathfrak{A}(a) \cdot \vec{1}) = \mathfrak{A}(a)$. \square

We now consider the means associated with the τ -function. In the following $\log = \log_e$ denotes the natural logarithm (to base e).

Definition 8.3.5. For $k \geq 2$ the map $\mathfrak{T}_k : \mathcal{BT}^{(k)} \rightarrow \mathbb{R}_{>0}$ is defined by

$$\mathfrak{T}_k(t) := \frac{\log(k)}{\log(\tau(t))} = \log_{\tau(t)}(k).$$

Note that while a smaller $\tau(t)$ indicates a better t , for the τ -mean \mathfrak{T} the larger $\mathfrak{T}_k(t)$ is the better t is, but this only holds for fixed k , and due to the normalisation the τ -means of tuples of different lengths cannot be compared.

Theorem 8.3.4. *Consider $k \geq 2$.*

1. \mathfrak{T}_k is a strictly concave homogeneous mean.
2. We have $\mathfrak{M}_{2-k}(t) \leq \mathfrak{T}_k(t) \leq \mathfrak{A}(t)$, with equalities iff t is constant. Especially we have $\mathfrak{G}(t) \leq \mathfrak{T}_2(t) \leq \mathfrak{A}(t)$.
3. For $k \geq 3$ it is \mathfrak{T}_k 0-dominated and ∞ -ignoring, while \mathfrak{T}_2 is 0-dominated as well as ∞ -dominated, whence only defined for $t \in \overline{\mathbb{R}}_{\geq 0}^2 \setminus \{(0, \inf), (\inf, 0)\}$.

In general we have that if a positive function $f : C \rightarrow \mathbb{R}_{>0}$ is “reciprocal-concave”, that is, $1/f$ is concave, then f is log-convex (but not vice versa), that is, $\log \circ f$ is convex. Furthermore, if f is log-convex then f is convex (but not vice versa).

Corollary 8.3.5. *Consider $k \geq 2$.*

1. τ_k is strictly reciprocal-log-concave for $k \geq 2$, that is, the map $t \in \mathcal{BT}^{(k)} \mapsto 1/\log(\tau(t)) \in \mathbb{R}_{>0}$ is strictly concave. Thus τ is strictly log-log-convex.
2. For $t \in \mathcal{BT}^{(k)}$ we have $\tau(\mathfrak{A}(t) \cdot \vec{1}) = k^{1/\mathfrak{A}(t)} \leq \tau(t) \leq k^{1/\mathfrak{M}_{2-k}(t)} = \tau(\mathfrak{M}_{2-k}(t) \cdot \vec{1})$.

Remarks:

1. Part 2 says that replacing all entries in a branching tuple by the arithmetic mean of the branching tuple improves (decreases) the τ -value, while replacing all entries by the geometric mean ($k = 2$) resp. harmonic mean

($k = 3$) impairs (increases) the τ -value. The case $k = 2$ of this inequality was shown in [KL98] (Lemma 5.6). For the use of the lower and upper bounds in heuristics see Subsection 8.6.

2. Computation of $\tau(t)$ can only be accomplished numerically (except of a few special cases), and a suitable method is the Newton-method (for computing the root of $\chi(t)(x) - 1$), where using the lower bound $|t|^{1/\mathfrak{A}(t)}$ as initial value performs very well (guaranteeing monotonic convergence to $\tau(t)$).

8.3.4. Associating probability distributions with branching tuples

Definition 8.3.6. Given a branching tuple $a = (a_1, \dots, a_k)$, the branching tuple $\tau^P(a) \in \mathcal{BT}^{(k)}$ is defined by $\tau^P(a)_i := \tau(a)^{-a_i}$ for $i \in \{1, \dots, k\}$.

Remarks:

1. By definition we have $\Sigma(\tau^P(a)) = 1$, and thus $\tau^P(a)$ represents a probability distribution (on $\{1, \dots, k\}$).
2. For $\lambda \in \mathbb{R}_{>0}$ we have $\tau^P(\lambda \cdot a) = \tau^P(a)$, and thus $\tau^P(a)$ only depends on the relative sizes of the entries a_i , and not on their absolute sizes.
3. For fixed k we have the map $\tau_k^P : \mathcal{BT}^{(k)} \rightarrow \text{int}(\sigma_{k-1}) \subset \mathcal{BT}^{(k)}$ from the set of branching tuples (a_1, \dots, a_k) of length k to the interior of the $(k - 1)$ -dimensional standard simplex σ_{k-1} , that is, to the set of all branching tuples $(p_1, \dots, p_k) \in \mathcal{BT}^{(k)}$ with $p_1 + \dots + p_k = 1$. We have already seen that $\tau_k^P(\lambda \cdot a) = \tau_k^P(a)$ holds. Furthermore τ_k^P is surjective, i.e., every probability distribution of size k with only nonzero probabilities is obtained, with $(\tau_k^P)^{-1}((p_1, \dots, p_k)) = \mathbb{R}_{>0} \cdot (-\log(p_1), \dots, -\log(p_k))$.

8.4. Estimating tree sizes

Now we turn to the main application of branching tuples and the τ -function, the estimation of tree sizes. We consider rooted trees (T, r) , where T is an acyclic connected (undirected) graph and r , the root, is a distinguished vertex of T . Since we are considering only rooted trees here, we speak in the sequel just of “trees” T with root $\text{rt}(T)$ and vertex set $V(T)$. We use $\#\text{nds}(T) := |V(T)|$ for the number of nodes of T , while $\#\text{lvs}(T) := |\text{lvs}(T)|$ denotes the number of leaves of T .

8.4.1. Notions for trees

For a node $v \in V(T)$ let $d_T(v)$ be the *depth* of v (the length of the path from the root to v), and for $i \in \{0, \dots, d(v)\}$ let $T(i, v)$ be the vertex with depth i on the path from the root to v (so that $T(0, v) = \text{rt}(T)$ and $T(d(v), v) = v$). For a node $v \in V(T)$ we denote by T_v the subtree of T with root v . Before turning to our main subject, measuring the size of trees, some basic strategic ideas should be pointed out:

- Trees are considered as “static”, that is, as given, not as evolving; the main advantage of this position is that it enables us to consider the “real”

backtracking trees, in contrast to the standard method of ignoring the real object and only to consider approximations given by recursion equations.

- The number of leaves is a measure easier to handle than the number of nodes: When combining trees under a new root, the number of leaves behaves additively, while the number of nodes is bigger by one node (the new root) than the sum. Reductions, which correspond to nodes with only a single successor, are being ignored in this way. For binary trees (every inner node as exactly two children) we have $\#nds(T) = 2\#lvs(T) - 1$. And finally, heuristics in a SAT solver aim at reducing the number of conflicts found, that is, the number of leaves.
- All leaves are treated equal (again, this corresponds to the point of view of the heuristics).

8.4.2. Adorning trees with probability distributions

Consider a finite probability space (Ω, P) , i.e., a finite set Ω of “outcomes” together with a probability assignment $P : \Omega \rightarrow [0, 1]$ such that $\sum_{\omega \in \Omega} P(\omega) = 1$; we assume furthermore that no element has zero probability ($\forall \omega \in \Omega : P(\omega) > 0$). The random variable P^{-1} on the probability space (Ω, P) assigns to every outcome ω the value $P(\omega)^{-1}$, and a trivial calculation shows that the expected value of P is the number of outcomes:

$$E(P^{-1}) = \sum_{\omega \in \Omega} P(\omega)P(\omega)^{-1} = |\Omega|. \quad (8.1)$$

So the random variable P^{-1} associates to every outcome ω a guess $P^{-1}(\omega)$ on the (total) number of outcomes, and the expected value of these guesses is the true total number of all outcomes. Thus, via sampling of P^{-1} we obtain an estimation on $|\Omega|$. In the general context this seems absurd, since the probabilities of outcomes are normally not given a priori, however in our application, where the outcomes of the probability experiment are the leaves of the search tree, we have natural ways at hand to calculate for each outcome its probability. We remark that for $r \in \mathbb{R}_{\geq 1}$ from (8.1) we get for the r -th moment the lower bound $E((P^{-1})^r) = E((P^{-r})) \geq |\Omega|^r$ (by Jensen’s inequality).

Definition 8.4.1. For trees T we consider *tree probability distributions* \mathfrak{P} , which assign to every edge (v, w) in T a probability $\mathfrak{P}((v, w)) \in [0, 1]$ such that for all inner nodes v we have $\sum_{w \in ds_T(v)} \mathfrak{P}((v, w)) = 1$, that is, the sum of outgoing probabilities is 1; we assume furthermore, that no edge gets a zero probability. We obtain the associated probability space (Ω_T, P) , where $\Omega_T := lvs(T)$, that is, the outcomes are the leaves of T , which have probability

$$P(v) := \prod_{i=0}^{d(v)-1} \mathfrak{P}((T(i, v), T(i+1, v))). \quad (8.2)$$

The edge-probabilities being non-zero just means that no outcome in this probability space has zero probability (which would mean it would be ignored).

Equation (8.2) makes sense for any vertex $v \in V(T)$, and $P(v)$ is then to be interpreted as the event that an outcome is a leaf in the subtree of T with root v (that is, $P(v) = \sum_{w \in \text{lvs}(T_v)} P_T(w)$); however we emphasise that the values $P(v)$ for inner nodes v are only auxiliary values. From (8.1) we obtain:

Lemma 8.4.1. *For every finite rooted tree T and every tree probability distribution \mathfrak{P} for T we have for the associated probability space Ω_T and the random variable $P^{-1} : \Omega_T \rightarrow]0, 1]$:*

$$\min P^{-1} = \min_{v \in \text{lvs}(T)} P(v)^{-1} \leq \#\text{lvs}(T) = E(P^{-1}) \leq \max_{v \in \text{lvs}(T)} P(v)^{-1} = \max P^{-1}.$$

Corollary 8.4.2. *Under the assumptions of Lemma 8.4.1 the following assertions are equivalent:*

1. $\min P^{-1} = \#\text{lvs}(T)$.
2. $\#\text{lvs}(T) = \max P^{-1}$.
3. P is a uniform distribution (all leaves have the same probability).

Lemma 8.4.1 opens up the following possibilities for estimating the size of a tree T , given a tree probability distribution \mathfrak{P} :

1. Upper bounding $\max P^{-1}$ we obtain an upper bound on $\#\text{lvs}(T)$, while lower bounding $\min P^{-1}$ we obtain a lower bound on $\#\text{lvs}(T)$.
2. Estimating $E(P^{-1})$ by sampling we obtain an estimation of $\#\text{lvs}(T)$.

By Corollary 8.4.2, in each case a tree probability distribution \mathfrak{P} yielding a uniform distribution p on the leaves is the most desirable distribution (the lower and upper bounds coincide with the true value, and only one path needs to be sampled). It is easy to see that each tree has exactly one such “optimal tree probability distribution”:

Lemma 8.4.3. *Every finite rooted tree T has exactly one tree probability distribution \mathfrak{P} which induces a uniform probability distribution P on the leaves, and this canonical tree probability distribution \mathfrak{CP}_T is given by*

$$\mathfrak{CP}_T((v, w)) = \frac{\#\text{lvs}(T_w)}{\#\text{lvs}(T_v)}$$

for $v \in V(T)$ and $w \in \text{ds}_T(v)$.

The canonical tree probability distribution \mathfrak{CP}_{T_v} on a subtree T_v of T (for $v \in V(T)$) is simply obtained by restricting \mathfrak{CP}_T to the edges of T_v (without change).

8.4.3. The variance of the estimation of the number of leaves

If the leaf probabilities vary strongly, then the variance of the random variable P^{-1} will be very high, and a large number of samples is needed to obtain a reasonable estimate on the number of leaves. So we should consider more closely the variance

$\text{Var}(P^{-1}) = E((P^{-1} - \#\text{lvs}(T))^2) = E(P^{-2}) - \#\text{lvs}(T)^2 \in \mathbb{R}_{\geq 0}$ of the random variable P^{-1} . By definition, the variance is 0 if and only if the probability distribution is uniform, that is, iff \mathfrak{P} is the canonical tree probability distribution on T . To estimate $\text{Var}(P^{-1})$ one needs to estimate $E(P^{-2})$, that is, the second moment of P^{-1} . By definition we have $E(P^{-2}) = \sum_{v \in \text{lvs}(T)} P(v) \cdot P(v)^{-2} = \sum_{v \in \text{lvs}(T)} P(v)^{-1}$. So $E(P^{-2})$ is just the sum over all estimations on $\#\text{lvs}(T)$ we obtain from the probability distribution P . Somewhat more efficiently, we can calculate all moments of P^{-1} recursively (using a number of arithmetical operations which is linear in $\#\text{nds}(T)$) as follows, where we use \mathfrak{P}_{T_v} for the restriction of the tree probability distribution $\mathfrak{P} = \mathfrak{P}_T$ to subtree T_v (unchanged), while P_{T_v} is the probability distribution induced by \mathfrak{P}_{T_v} (which is not the restriction of P_T ; for $w \in \text{lvs}(T_v)$ we have $P_{T_v}(w) = P_T(v)^{-1} \cdot P_T(w)$). Trivial calculations show:

Lemma 8.4.4. *For a finite rooted tree T , a tree probability distribution \mathfrak{P} on T and $r \in \mathbb{R}_{\geq 0}$ we can recursively compute the r -th moment $E(P^{-r})$ of P^{-1} by*

- If T is trivial (i.e., $\#\text{nds}(T) = 1$), then we have $E(P_T^{-r}) = 1$.
- Otherwise $E(P_T^{-r}) = \sum_{v \in \text{ds}(\text{rt}(T))} \mathfrak{P}_T(\text{rt}(T), v)^{1-r} \cdot E(P_{T_v}^{-r})$.

Following [Knu75] (Theorem 3), we give an estimation on the variance of P^{-1} if the derived transition probabilities on the edges are within a factor α of the canonical tree probability distribution (recall Lemma 8.4.3).

Lemma 8.4.5. *For a finite rooted tree T and a tree probability distribution \mathfrak{P} on T , which fulfils $\mathfrak{P}_T^{-1} \leq \alpha \cdot \mathfrak{C}\mathfrak{P}_T^{-1}$ for some $\alpha \in \mathbb{R}_{\geq 1}$, we have $E(P_T^{-r}) \leq \alpha^{(r-1) \cdot \text{ht}(T)} \cdot \#\text{lvs}(T)^r$ for all $r \in \mathbb{R}_{\geq 1}$.*

Corollary 8.4.6. *Under the same assumptions as in Lemma 8.4.5 we have*

$$\text{Var}(P_T^{-1}) \leq (\alpha^{\text{ht}(T)} - 1) \cdot \#\text{lvs}(T)^2.$$

All the considerations of this section can be generalised to the case, where we also take inner nodes of the tree into account, and where we have an arbitrary cost function which assigns to every node of the tree its cost (in this section we considered the cost function which assigns every leaf the cost 1, while every inner node gets assigned cost 0). See [Kul08a] for details.

8.4.4. The tau-method

In the previous subsection we developed a method of estimating tree sizes, assuming a given tree probability distribution which assigns to every edge a transition probability. Now in this subsection we discuss how to obtain such transition probabilities via the help of “distances” and “measures”. The basic idea is to attach a distance $d((u, v))$ to the edge (u, v) , measuring how much “progress” was achieved via the transition from u to v , and where a standard method for obtaining such distances is to use a measure μ of “complexity” via $d((u, v)) := \mu(u) - \mu(v)$.

Definition 8.4.2. A *distance* d on a finite rooted tree T is a map $d : E(T) \rightarrow \mathbb{R}_{>0}$ which assigns to every edge (v, w) in T a positive real number, while a *measure* is a map $\mu : V(T) \rightarrow \mathbb{R}$ such that $\Delta\mu$ is a distance, where $\Delta\mu((v, w)) := \mu(v) - \mu(w)$. For a distance d we define the measures $\min \Sigma d$ and $\max \Sigma d$ on T , which assign to every vertex $v \in V(T)$ the minimal resp. maximal sum of d -distances over all paths from v to some leaf.

The τ -function yields a canonical method to associate a tree probability distribution to a distance on a tree as follows.

Definition 8.4.3. Consider a rooted tree T together with a distance d . For an inner node v of T we obtain an associated branching tuple $d(v)$ modulo permutation; assuming that T is ordered, i.e., we have a sorting $\text{ds}_T(v) = \{v_1, \dots, v_k\}$, we obtain a concrete branching tuple $d(v) := (d(v, v_1), \dots, d(v, v_k))$. The associated tree probability distribution \mathfrak{P}_d is given by

$$\mathfrak{P}_d((v, v_i)) := \tau^P(d(v))_i$$

(recall Definition 8.3.6).

By definition we have for $\lambda \in \mathbb{R}_{>0}$ that $\mathfrak{P}_{\lambda \cdot d} = \mathfrak{P}_d$. By Remark 3 to Definition 8.3.6 for every tree probability distribution \mathfrak{P} for T there exist distances d on T with $\mathfrak{P} = \mathfrak{P}_d$, and d is unique up to scaling of the branching tuples at each inner node (but each inner node can be scaled differently).

Given a distance d on a tree T , in order to apply Lemma 8.4.1 we need to estimate $\min P_d^{-1}$ and $\max P_d^{-1}$, where P_d is the probability distribution induced by \mathfrak{P}_d on the leaves of T according to Definition 8.4.1.

Definition 8.4.4. For a rooted tree (T, r) with a distance d let $\min \tau(d) := +\infty$ and $\max \tau(d) := 1$ in case T is trivial (consists just of r), while otherwise $\min \tau(d) := \min_{v \in V(T) \setminus \text{lvs}(T)} \tau(d(v))$ and $\max \tau(d) := \max_{v \in V(T) \setminus \text{lvs}(T)} \tau(d(v))$.

Lemma 8.4.7. Consider a rooted tree (T, r) together with a distance d . For the induced probability distribution P_d on the leaves of T we have:

1. $(\min \tau(d))^{\min \Sigma d(r)} \leq \min P_d^{-1}$.
2. $\max P_d^{-1} \leq (\max \tau(d))^{\max \Sigma d(r)}$.

Proof. We prove Part 1 (the proof for Part 2 is analogous). If T is trivial then the assertion is trivial, so assume that T is non-trivial. Let $\tau_0 := \min \tau(d)$.

$$\begin{aligned} \min P_d^{-1} &= \min_{v \in \text{lvs}(T)} P_d(v)^{-1} = \min_{v \in \text{lvs}(T)} \prod_{i=0}^{d(v)-1} \mathfrak{P}_d(T(i, v), T(i+1, v))^{-1} = \\ &= \min_{v \in \text{lvs}(T)} \prod_{i=0}^{d(v)-1} \tau(d(T(i, v)))^{d(T(i, v), T(i+1, v))} \leq \\ &= \min_{v \in \text{lvs}(T)} \prod_{i=0}^{d(v)-1} \tau_0^{d(T(i, v), T(i+1, v))} = \min_{v \in \text{lvs}(T)} \tau_0^{\sum_{i=0}^{d(v)-1} d(T(i, v), T(i+1, v))} = \tau_0^{\min \Sigma d(r)}. \end{aligned}$$

□

Lemma 8.4.1 together with Lemma 8.4.7 yields immediately the following fundamental method for estimating tree sizes.

Theorem 8.4.8. *Consider a rooted tree (T, r) . For a distance d on (T, r) we have*

$$(\min \tau(d))^{\min \Sigma d(r)} \leq \#\text{lvs}(T) \leq (\max \tau(d))^{\max \Sigma d(r)}.$$

And for a measure μ on (T, r) which is 0 on the leaves we have

$$(\min \tau(\Delta\mu))^{\mu(r)} \leq \#\text{lvs}(T) \leq (\max \tau(\Delta\mu))^{\mu(r)}.$$

Remarks:

1. So upper bounds on tree sizes are obtained by Theorem 8.4.8 through

- upper bounds on the τ -values on the branching tuples at each inner node of the tree
- and upper bounds on the maximal sum of distances amongst paths in the tree,

where the latter can be obtained via the root-measure in case the distances are measure-differences.

2. The general method of Theorem 8.4.8 was introduced by [Kul99b] (Lemma 8.2 there; see Section 8.7.3 here for more on the topic of theoretical upper bounds), while in [KL97, KL98] one finds direct (inductive) proofs.

The ideal measure on a tree makes the bounds from Theorem 8.4.8 becoming equal:

Lemma 8.4.9. *Consider a non-trivial rooted tree (T, r) . Then the following assertions are equivalent for a measure μ on (T, r) which is 0 on the leaves:*

1. *There exists $\lambda \in \mathbb{R}_{>0}$ with $\forall v \in V(T) : \mu = \lambda \cdot \log(\#\text{lvs}(T_v))$.*
2. $(\min \tau(\Delta\mu))^{\mu(r)} = \#\text{lvs}(T)$.
3. $(\max \tau(\Delta\mu))^{\mu(r)} = \#\text{lvs}(T)$.
4. $\min \tau(\Delta\mu) = \max \tau(\Delta\mu)$.

If one of these (equivalent) conditions are fulfilled then $\mathfrak{P}_{\Delta\mu} = \mathfrak{CP}_T$ (the canonical tree probability distribution; recall Lemma 8.4.3).

So a good distance d on a rooted tree T has to achieve two goals:

1. Locally, for each inner node v with direct successors $\text{ds}(v) = \{v_1, \dots, v_k\}$ the relative proportions of the distances $d(v, v_i)$ must mirror the sizes of the subtrees hanging at v_i (the larger the subtree the smaller the distance).
2. Globally, the scaling of the different τ -values at inner nodes must be similar.

As we have seen in Lemma 8.4.9, if the second condition is fulfilled perfectly, then also the first condition is fulfilled (perfectly), while the other direction does not hold (the first condition can be achieved with arbitrary scaling of single branching

tuples). The problem of constructing good distance functions (from a general point of view) is further discussed in Section 8.8.

After having seen that the τ -function can be put to good use, in the following section we show that at least the linear quasi-order induced on branching tuples by the τ -function (the smaller the τ -value the “better” the tuple) follows from very fundamental requirements on the evaluation of branchings.

8.5. Axiomatising the canonical order on branching tuples

On \mathcal{BT} we have a natural linear quasi-order given by $a \leq b \Leftrightarrow \tau(a) \leq \tau(b)$ for $a, b \in \mathcal{BT}$. Here we show how this order follows from simple intuitive axioms (extending [Kul98]).

Definition 8.5.1. A relation \leq on \mathcal{BT} is called a **canonical branching order** if it fulfils the six properties (TQO), (CMP), (P), (W), (M) and (Con), which are the following conditions (for all branching tuples a, b, c and all permutations π), where the induced equivalence relation $a \sim b :\Leftrightarrow a \leq b \wedge b \leq a$ and the induced strict order $a < b :\Leftrightarrow a \leq b \wedge a \not\sim b$ are used. First the five elementary conditions are stated:

- (TQO) (“Total Quasi-order”) \leq is a total quasi-order on \mathcal{BT} , that is, $a \leq a$, $a \leq b \wedge b \leq c \Rightarrow a \leq c$ and $a \leq b \vee b \leq a$ for all branching tuples a, b, c .
- (CMP) (“Composition”) $a \leq b \Rightarrow (a \leq a \wedge b \leq b) \wedge (a \leq b \wedge a \leq b)$.
- (P) (“Permutation”) $\pi * a \sim a$.
- (W) (“Widening”) $a \sqsubset b \Rightarrow a < b$.
- (M) (“Monotonicity”) If $k := |a| = |b| \geq 2$, $\forall i \in \{1, \dots, k\} : a_i \leq b_i$ and $\exists i \in \{1, \dots, k\} : a_i > b_i$, then $a < b$.

Now for $a \in \mathcal{BT}$ let $\gamma_a : \mathbb{R}_{>0} \rightarrow \mathcal{BT}$ be defined by $\gamma_a(x) := a;(x)$ (“left translation”). So (M) just expresses that for $a \in \mathcal{BT}$ the map γ_a is strictly decreasing. And extend $\gamma_a : \mathbb{R}_{>0} \rightarrow \mathcal{BT}$ to $\gamma_a : \overline{\mathbb{R}}_{>0} \rightarrow \mathcal{BT}$ by $\gamma(+\infty) := a$. The remaining condition is:

- (Con) (“Continuity”) For $a \in \mathcal{BT}$ the map $\gamma_a : \overline{\mathbb{R}}_{>0} \rightarrow \mathcal{BT}$ is continuous with regard to the natural topology on $\overline{\mathbb{R}}_{>0}$ and the order topology on \mathcal{BT} , i.e.:
 - (Con1) For $x \in \mathbb{R}_{>0}$ and $b, c \in \mathcal{BT}$ with $b < \gamma_a(x) < c$ there is $\delta \in \mathbb{R}_{>0}$ such that for $x' \in \mathbb{R}_{>0}$ with $|x - x'| < \delta$ we have $b < \gamma_a(x') < c$.
 - (Con2) For $b \in \mathcal{BT}$ with $b > a$ there is $x_0 \in \mathbb{R}_{>0}$ such that for $x' \in \mathbb{R}_{>x_0}$ we have $b > \gamma_a(x')$.

Remarks:

1. The intuitive meaning of “ $a \leq b$ ” is that “in general”, that is, “if nothing else is known”, branching tuple a is at least as good as b (doesn’t lead to larger branching trees). The main result of this section is that actually there is exactly one canonical branching order.

2. (TQO) expresses the comparability of branching tuples; the order does not fulfil antisymmetry (i.e., $a \leq b \wedge b \leq a \Rightarrow a = b$), since for example, as stated in (P), permutation doesn't change the value of a branching, and via (CMP) also composition of a branching tuple with itself doesn't change its value.
3. (CMP) states that if a is at least as good as b , then $a \mathbb{M} b$ as well as $b \mathbb{M} a$ are "compromises", improving b and impairing a .
4. (P) says permutation does not change anything essential.
5. (W) requires that adding branches to a branching impairs the branching.
6. (M) states that increasing some component of a branching tuple of width at least two strictly improves the branching tuple.
7. Finally (Con) states that sufficiently small changes in one component yield only small changes in the "value" of the tuple.

Lemma 8.5.1. *The linear quasi-order on \mathcal{BT} given by $a \leq b \Leftrightarrow \tau(a) \leq \tau(b)$ for $a, b \in \mathcal{BT}$ is a canonical branching order.*

Lemma 8.5.2. *Consider a canonical branching order \leq on \mathcal{BT} . Then for $a, b \in \mathcal{BT}$ with $\tau(a) < \tau(b)$ we have $a < b$.*

Theorem 8.5.3. *There is exactly one canonical branching order on branching tuples, given by $a \leq b \Leftrightarrow \tau(a) \leq \tau(b)$ for all $a, b \in \mathcal{BT}$.*

Proof. Consider a canonical branching order \leq . We have to show that for all $a, b \in \mathcal{BT}$ we have $a \leq b \Leftrightarrow \tau(a) \leq \tau(b)$. By Lemma 8.5.1 we know the direction from right to left. So assume that \leq is a canonical branching order, and consider $a, b \in \mathcal{BT}$. If $a \leq b$ holds, then $\tau(a) > \tau(b)$ by Lemma 8.5.2 would imply $a > b$ contradicting the assumption. So assume now (finally) $\tau(a) \leq \tau(b)$, and we have to show that $a \leq b$ holds. If $a > b$ would be the case, then by (Con1) there exists $\varepsilon \in \mathbb{R}_{>0}$ with $a \mathbb{M}(\varepsilon) > b$, however we have $\tau(a \mathbb{M}(\varepsilon)) < \tau(a) \leq \tau(b)$, and thus by Lemma 8.5.2 it would hold $a \mathbb{M}(\varepsilon) < b$. \square

For branching tuples with rational entries the canonical branching order can be decided in polynomial time (in the binary representation) by using the decidability of the first-order theory of the real numbers ([Ren92]). The approach of this section can be generalised by considering only tuples of length at most k resp. of length equal k , and formulating the axioms in such a way that all occurring branching tuples are in the restricted set. In the light of the questions arising in the subsequent Section 8.6 about projections for branching tuples of restricted width such generalisations seem to be worth to study.

8.6. Alternative projections for restricted branching width

The argumentation of Subsection 8.5 depends on considering branching tuples of arbitrary length. The strength of the τ -function is that it imposes a consistent scaling for tuples of different sizes, and Theorem 8.5.3 shows that the order induced by the τ -function (the canonical order) is the only reasonable order if

branching tuples of arbitrary width are considered. Now if we consider only branching tuples of some constant length, then other choices are possible. Practical experience has shown that for a binary branching tuple (a_1, a_2) maximising the product $a_1 \cdot a_2$ yields good results, and this projection is universally used now (ignoring tie-breaking aspects here), while maximising the sum $a_1 + a_2$ has been shown by all experiments to perform badly. We are now in a position to give theoretically founded explanations:

1. The general rule is that $\tau(a_1, a_2)$ should be minimised.
2. If computation of $\tau(a_1, a_2)$ is considered to be too expensive, then the approximations from Corollary 1, Part 2 could be used, which amount here to either maximise the arithmetic mean $\mathfrak{A}(a_1, a_2)$ or to maximise the geometric mean $\mathfrak{G}(a_1, a_2)$. Now maximising $\mathfrak{A}(a_1, a_2)$ is equivalent to maximising the sum $a_1 + a_2$, while maximising the geometric mean $\mathfrak{G}(a_1, a_2)$ is equivalent to maximising the product $a_1 \cdot a_2$.
3. So maximising the sum $a_1 + a_2$ means to minimise a lower bound on the τ -value, while maximising the product $a_1 \cdot a_2$ means minimising an upper bound on the τ -value — it appears now that the second choice is more meaningful, since it amounts to minimise an upper bound on the tree size, while minimising a lower bound on the tree size leads to nowhere.

A more quantitative explanation is given by the following lemma, which shows that the product yields a better approximation to the τ -mean than the sum.

Lemma 8.6.1. *We have $\mathfrak{A}(a_1, a_2) - \mathfrak{T}(a_1, a_2) \geq \mathfrak{T}(a_1, a_2) - \mathfrak{G}(a_1, a_2)$ for all $a_1, a_2 \in \mathbb{R}_{>0}$, with equality iff $a_1 = a_2$ (and in this case both sides are zero).*

Especially for branching tuples with more than two entries the τ -function appears as the canonical choice; if approximations are sought then the product (corresponding to the geometrical mean) can no longer be used, but the general upper bound from Corollary 1, Part 2 is a candidate.³ It is not clear whether for branching tuples of constant width the canonical order is always the superior choice (and using for example the product-rule for binary branches is just a reasonable approximation), or whether there might be special (but still “general”) circumstances under which other orders are preferable. Considering again binary branchings, for the canonical order the branching tuples $(1, 5)$ and $(2, 3)$ are equivalent, but they are distinguished by the product rule which favours $(2, 3)$. This is in fact a general property of the product rule (which, as already said, is only sensible for binary tuples) that it favours more symmetric tuples (compared to the canonical order).

³The unsuitability of the product for branching widths at least 3 can be seen for example by the fact that the product is ∞ -dominated (i.e., if one component goes to infinity, so does the product), while the τ -function is ∞ -ignorant — if one branch is very good, then this does not mean that the whole branching is very good, but only that this branch doesn’t contribute to the overall cost.

8.7. How to select distances and measures

After having laid the foundations, in this section we consider now the basic “branching schemes” used by such (practical) SAT solvers employing branching algorithms. The essential ingredient of these “schemes” (made more precise in Subsection 8.7.2) is the underlying distance function, however obviously the whole of the algorithm is important, and the aspects related to the branching heuristics are condensed in these “schemes”.⁴ The main purpose is to give an overview on the relevant measures $\mu(F)$ of “absolute problem complexity” or of distances $d(F, F')$ as measurements of “relative problem complexity”, as core ingredients of the heuristical melange found in a SAT solver. This is intertwined with especially the reduction process, and we discuss this aspect further in Subsection 8.7.2. A potential source for future heuristics is the literature on worst-case upper bounds, and we make some comments in Subsection 8.7.3. Then in Subsection 8.7.4 we give a “rational reconstruction” of the branching schemes currently used in practice, which actually all can be explained as “maximising the number of new clauses” — this branching rule in its pure form was implemented by the `OKsolver-2002` (see [Kul02]), and is presented in Subsection 8.7.4.1, while the historical development in practical SAT solving leading to this rule is discussed in Subsection 8.7.4.2. For background on “look-ahead SAT solvers” in general see Chapter 5 of this handbook.

8.7.1. Some basic notations for clause-sets

Applying the abstract ideas and notions to concrete problems, now we need to consider more closely actual problem instances. In the sequel we use the following notations for (multi-)clause-sets F :

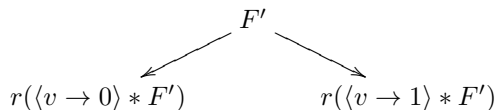
- $n(F) := |\text{var}(F)|$ is the number of variables (actually occurring in F).
- $c(F) := |F|$ is the number of clauses, and more specifically $c_k(F)$ for $k \in \mathbb{N}_0$ is the number of clauses of length (exactly) k .
- $\ell(F) = \sum_{C \in F} |C| = \sum_{k \in \mathbb{N}_0} c_k(F) \cdot k$ is the number of literal occurrences.
- The maximal clause-length is denoted by $\text{rank}(F)$.
- For a partial assignment φ the result of applying φ to F is denoted by $\varphi * F$ (eliminating satisfied clauses, and removing falsified literals from the remaining clauses).
- The partial assignment (just) assigning value ε to variable v is denoted by $\langle v \rightarrow \varepsilon \rangle$.
- $r_k(F)$ denotes generalised unit-clause propagation (see [Kul99a]), i.e., r_0 just reduces F to $\{\perp\}$ iff $\perp \in F$, r_1 is unit-clause propagation, r_2 is failed-literal propagation etc.

⁴As remarked earlier, yet heuristics for conflict-driven solvers are mainly “purely heuristical”, and no theoretical foundation exists, whence we do not consider them here. Some general remarks on heuristics in this context are in [Kul08b]. From a theoretical angle, in [BKS04] guidance by structure has been considered (on an example), while [BKD⁺04] considered the relation between tree decompositions and learning.

8.7.2. On the notion of “look-ahead”

An important concept in this context is the notion of “look-ahead”, which is used in the literature in different ways, and so needs some discussion. A *reduction* is typically a function $r(F)$ for problem instances F , computable in polynomial time such that problem instance $r(F)$ is satisfiability equivalent to F . Now for example in [Fre95] “look-ahead” is used mainly instead of “reduction” (actually framed in the language of constraint satisfaction, as “constraint propagator”), and this reduction process can actually be integrated into the heuristics for choosing the branching variable. This integration of reduction and heuristics seems to be rather popular amongst current look-ahead architectures, however in this article, with its foundational emphasis, we need clean and precise distinctions, and so we use the following architecture of a recursive backtracking SAT solver:

1. The input is a problem instance F .
2. Via the *reduction* r the instance is reduced to $F' := r(F)$, including at least unit-clause propagation (iterated elimination of (all) unit-clauses).
3. If now F' is trivially satisfiable or unsatisfiable, as established by the *immediate-decision oracle*, then the result is returned (more complex tests for satisfiability or unsatisfiability can be integrated into the reduction).
4. The purpose of the *branching scheme* (for standard SAT solvers) is now to select a *branching variable* $v \in \text{var}(F')$ such that branching using the two branches



yields the “fastest” decision (applying the algorithm recursively to the branches). Note that from this point of view branching considers only reduced instances. For sequential execution the ordering of the branches is important, and is accomplished by the *branching ordering heuristics*. The tasks of the branching scheme and the branching ordering heuristics are accomplished as follows:

- (a) For each variable $v \in \text{var}(F')$ and value $\varepsilon \in \{0, 1\}$ the *look-ahead reduction* r' computes an approximation of $r(\langle v \rightarrow \varepsilon \rangle * F')$ as $F_v^\varepsilon := r'(\langle v \rightarrow \varepsilon \rangle * F')$, where r' is a weaker form of r .
- (b) Via the *distance function* d for each variable v the branching tuple $t_v := (d(F', F_v^0), d(F', F_v^1)) \in \mathbb{R}_{>0}^2$ is computed.
- (c) Via the *projection* ρ each branching tuple t_v is projected to a single real number $\rho(t_v)$, and the branching scheme selects a variable v_0 with minimal $\rho(t_{v_0})$ (possibly using additional distance functions to break ties).
- (d) The branching order finally is established by a *satisfiability estimator* $P(F_{v_0}^\varepsilon)$, a function which computes for each branch $F_{v_0}^\varepsilon$ an “approximation” of the chance of being satisfiable, and the branch with higher P -value is chosen first.

In order to separate reduction and branching scheme we require here that no reduction takes place in Step 4 (where the branching scheme performs its work), that is, no F_v^ε is trivially decidable (according to Step 3), and thus r must actually be stronger than r' . Often r is the “look-ahead version” of r' , that is, if r' is the identity, then r is just unit-clause propagation, and if r' is unit-clause propagation then r is failed-literal elimination; in general, if $r' = r_k$ then often $r = r_{k+1}$. In practice typically the separation between reduction and heuristic is blurred, and the choice of the “branching heuristics” involves a mixture of choices for reduction, look-ahead reduction, distances, projections and satisfiability estimator, typically applied in an entangled way to improve efficiency, but for the correct understanding the above distinctions seem essential.

8.7.3. Branching schemes in theoretical algorithms

[Kul92, Kul99b] introduced the now dominant technique for proving worst-case upper bounds on NP-hard problems, called “measure-and-conquer” in [FGK05], which is based on Theorem 8.4.8 and the use of an appropriate “interesting” distance d or measure μ . For a recent work on upper bounds and measures see [Wah07], and Chapter 16 of this handbook for worst-case upper bounds in general, while in this chapter we only consider the particular strand of work in this area which influenced heuristics for SAT solving.

In [KL97, KL98] the basic measures n , c , ℓ for SAT decision have been considered for the first time systematically. [Kul99b], motivated by [Sch92], based his analysis on the (measure-based) distance $d^2 = \Delta m_k = \Delta n - \alpha \Delta z_k$, where z_k is a capped version of c_2 (in order to make c_2 and n comparable), and $\alpha \in \mathbb{R}_{>0}$ is to be optimised. Note that a *decrease* in n is favourable, while an *increase* in the number of binary clauses is favourable. An important refinement then replaces the distance d^2 by the distance d^3 , which takes (depending on the parameter k) a fixed amount of *new* binary clauses into account (as improvement), while “ignoring” the number of eliminated binary clauses. Starting from this measure, by extensive experimentation the heuristic for the `OKsolver-2002` was empirically “derived” ([Kul98]), arriving for 3-CNF at a distance function without magic numbers, which is just the number of *new binary clauses*.

That the distance Δn vanished finally in this development can be motivated as follows: Δn was used in [Kul99b] since an upper bound in the measure n was to be derived. But “heuristic” versions, which just care about making the tree smaller, without caring about accompanying (interpretable) upper bounds, can remove the factor n which imposed, due to the positivity requirement $d^3 > 0$, a cap on the number k of binary clauses to be taken into account — now arbitrary amounts of new binary clauses count! This branching scheme, generalised to arbitrary CNF will be further discussed in Subsection 8.7.4.1. It seems to be at this time the strongest “pure” heuristics known (for look-ahead solver, and used as basis for further heuristical extensions). And, by “magic coincidence”, its basic content is closely related to the emerging heuristics from practical SAT solving, as explained in Subsection 8.7.4.2.

8.7.4. Branching schemes in practical algorithms

Now we turn to branching schemes as proposed (and used) in the realm of “practical SAT solving”. First a few comments on very simple underlying distances, which might be useful for comparisons (or perhaps in the context of optimising combinations of distances, as considered in Section 8.8):

1. The trivial choice, an “arbitrary variable”, is reflected by choosing a constant distance function. Here the “tie-breaking rule” becomes decisive, and choosing a random variable, which for random instances is the same as choosing the first variable in a fixed (instance-independent) order, is considerably worse than choosing the first occurring variable in the clause-set, which for random instances in the average will choose the variable occurring most often.
2. Δn is perhaps the most basic non-trivial choice. This distance gets stronger with look-ahead, but for practical SAT solving it never played a role yet (different from the theoretical upper bounds as mentioned in Subsection 8.7.3). Δc and $\Delta \ell$ are more sensitive to the input⁵, but still these distances appear as not suitable for practical SAT solving in general. Likely the basic problem with these measures is that they do not provide any guidance towards increased efficiency of the reduction process (unit-clause propagation and beyond).

We present the branching scheme of the `OKsolver-2002` in Subsection 8.7.4.1 as the core rule: It represents a certain convergence of theoretical considerations coming from worst-case upper bounds (see Subsection 8.7.3) with practical developments (see Subsection 8.7.4.2), it has been deliberately kept free from purely heuristical considerations (as much as possible — only the clause-weights appear to be unavoidably of heuristical nature), and this one rule actually represents the core of modern branching rules for look-ahead solvers. As explained in Subsection 8.7.3, the underlying distance started from a combination of Δn and the number of new 2-clauses (for inputs in 3-CNF), but experimental results forced Δn to leave, and the result is close to the rule `dsj` as a result of the development starting with the Jeroslow-Wang rule — however the novelty is, and this came from the theoretical investigations in worst-case analysis, to view the whole process as applying a distance function, with the aim of maximising the number of *new clauses*.⁶ In the literature on practical SAT solving, this target (creating as many strong constraints as possible) has been first discussed in [Li99].

There have been a few attempts of going beyond the underlying distance:

1. In [Ouy99], Chapter 3, we find a scheme to dynamically adjust the weights of clauses of length 2, leading to branching rule “B”. However, since this considers the “old” clauses, it really means new clauses of length 1, i.e.,

⁵where for $c \geq n$ to hold, reduction by matching autarkies is needed (see [KL98, Kul03])

⁶The explanations in [HV95] for the first time concentrated on the role of unit-clause propagations, however from the point of view of simplifying the current formula, not, as the distance functions emphasises, from the point of view of the emergence of *future reductions*.

unit-clauses, which from our point of view shouldn't be considered in this form by the branching rule, but it should be considered by the (r_1) look-ahead. So this scheme seems only to be another approximation of the look-ahead, where at least for "look-ahead solvers" the r_1 -look-ahead seems mandatory now.

2. Van Maaren et al considered non-linear approximations of CNF in order to gain a deeper, geometrical understanding of the problem instance. These efforts culminated in the branchings rule discussed in [vW00], where branching rule "MAR" is presented. The main idea is to derive for the residual clause-set in a natural way an n -dimensional ellipsoid (where n is the number of variables of the current clause-set), where each variable corresponds to one axis of the ellipsoid, and then the geometry of the ellipsoid can tell us which variable might have the biggest influence. Again the problem is that it seems impossible to include into this picture the logical inferences, i.e., the look-ahead. The fundamental problem is that this point of view doesn't seem to deliver a measure or a distance, and thus it cannot be used to compare arbitrary problem instances. Perhaps this restriction can be overcome in the future.

The most promising direction at this time for strengthened distances is presented in Section 8.8, where, based on our general understanding of the interplay between distances and trees, possibilities are discussed to optimise distance functions, even online, so that one can start with a reasonable distance, as presented in the subsequent Subsection 8.7.4.1, and then adapt it to the problem at hand.

8.7.4.1. Maximise the number of new clauses

We start with presenting the current state-of-the-art, which in its pure form is the branching rule "MNC" ("maximise new clauses") used by the `OKsolver-2002`, and from this we make a "rational reconstruction" of earlier rules (in Subsection 8.7.4.2), which can well be understood as approximations of MNC.

MNC for a branching variable v and the residual clause-set F considers the two branches F_0, F_1 , where for a standard look-ahead solver we have $F_\varepsilon = \varphi_\varepsilon * F$, where φ_ε is the extension of $\langle v \rightarrow \varepsilon \rangle$ by unit-clause propagation, and uses as distance the weighted number of new clauses

$$d_{\text{MNC}}(F, F') = \sum_{k=2}^{\text{rank}(F)-1} w_k \cdot c_k(F' \setminus F).$$

More precisely, multi-clause-sets should be used here, since typically in practice multiple occurrences of the same new clause are counted, and no contraction with existing clauses takes place. By optimisation on random 3-CNF at the (approximated) threshold, optimal weights could be established as approximately $w_2 := 1$, $w_3 := 0.2$, $w_4 := 0.05$, $w_5 := 0.01$, $w_6 := 0.003$ and $w_k = 20.45 \cdot 0.2187^k$ for $k \geq 7$, where these weights also work quite well on other problem instances. The "precise" combination of the two distances $d_{\text{MNC}}(F, F_0), d_{\text{MNC}}(F, F_1)$ into

one number happens via the τ -function, while in practice the product-rule is sufficient (as discussed in Subsection 8.6). After the best variable has been chosen, the first branch is selected as will be discussed in Section 8.9, where in practical applications currently the scheme discussed in Subsection 8.9.1 seems to yield the best results. A few remarks:

1. We have $d_{\text{MNC}}(F, F_\varepsilon) = 0$ iff $F_\varepsilon \subseteq F$, in which case φ_ε is a “weak autarky” for F , and thus F_ε is satisfiability equivalent to F , so no branching is needed.
2. The more new clauses the better, and the better the shorter they are.
3. New clauses result from falsifying literal occurrences in (old) clauses which are not satisfied — satisfied clauses are ignored by the distance (but will be taken into account when choosing the first branch).
4. When performing look-ahead the behaviour of d_{MNC} seems counterintuitive: Consider two variables v, w , where for v we have in both branches many inferred assignments (by unit-clause propagation), but it happens that they both result in just, say, one new clause each; on the other hand, assume that for w in both branches there aren’t many inferred assignments (possibly none at all), but more new clauses. Then w will be preferred over v . Such examples can be constructed, but in practice it turned out that attempts at balancing d_{MNC} , by for example the number of inferred assignments, performed worse in almost all cases (which is also confirmed by the historical development, as outlined in the subsequent subsection).⁷

Instead of clauses also more general “conditions” (“constraints”) can be treated, if partial assignments can be applied to them, and instead of new clauses we then need to consider conditions whose domains have been restricted. Since conditions can show much more variety in their behaviour, the problem of the choice of weights for the “new conditions” becomes more pronounced; a first approximation is to replace the length k of clauses by the size of the domain of the condition, but in general one needs to keep in mind that the reason why shorter clauses are preferred over longer ones is that they are *more constrained*, i.e., will easier yield inferred assignments in the future. One can measure the ratio of falsifying assignments for the new conditions (the higher the better), but according to current knowledge extensive experimentation to find good weighting schemes seems unavoidable. One also needs to take into account that while clauses can only yield a single inferred assignment, other constraints can yields more inferred assignments (at once; and potentially also other types of information might be inferred, for example equality between variables). See Subsection 8.10.1 for some examples.

⁷The extreme case of zero new clauses needs to be handled (since distances need to be positive), and for this case we have the autarky-reduction above; in [Kul99b] also the case of an arbitrary number of new clauses is treated via the addition of “autarky clauses” (which actually goes beyond resolution), and the `OKsolver-2002` contains (deactivated) code for handling the case of exactly one new clause. The future has to show whether this scheme is of value.

8.7.4.2. The historical development (for practical SAT solving)

An early branching scheme is presented in [JW90], and is known as the “Jeroslow-Wang rule”. Since it is a rather confused rule, of weak efficiency and mixing up several aspects, we do not further comment on this scheme. However this scheme was at least historically of importance, since it allowed [HV95] to state an improved branching scheme, with improved (though still confused) reasoning: The Jeroslow-Wang rule is replaced by the “two-sided Jeroslow-Wang rule”, rejecting the idea that branching schemes are “satisfiability driven”, and replacing this paradigm by the “simplification paradigm”. We can interpret this rule by the underlying distance $d_{2JW}(F, F') := \sum_{k=1}^{\text{rank}(F)-1} w_k \cdot c_k(F' \setminus F)$ where $w_k := 2^{-k}$ (strictly following [HV95] it would be $w_k = 2^{-(k+1)}$, but obviously the factor $\frac{1}{2}$ doesn’t matter here). Reduction is unit-clause propagation and pure literal elimination. No look-ahead is used (fitting with the choice of reduction(!)), and as projection the sum is used. The choice of the first branch happens by the Johnson-rule (see Subsection 8.9.2). We see that d_{MNC} improves d_{2JW} by

1. using the product as projection;
2. discriminating sharper between different clause-length;
3. using r_1 -look-ahead (instead of r_0);
4. choosing of the first branch by the Franco-rule (see Subsection 8.9.1).

These shortcomings have been addressed piecewise by later developments as outlined below. Regarding our interpretation, we need to stress that the understanding of the “two-sided Jeroslow-Wang rule” as based on the distance of counting *new* clauses is a rational reconstruction, while the argumentation in [HV95] tries to argue in the direction of “simplification”. However this is not really what d_{2JW} is aiming at, namely increasing the number of future forced assignments (by unit-clause propagation or stronger means). In [HV95] we find a misleading understanding of the sum $d_{2JW}(F, \langle v \rightarrow 0 \rangle * F) + d_{2JW}(F, \langle v \rightarrow 1 \rangle * F)$ (maximised by the heuristics), which is there understood as estimating the simplification (as a kind of average over both branches), and so the real target of the rule, creating many new short clauses (for both branches(!)), where then the sum only acts as a (bad) projection, remained hidden.

[VT96] improves upon this branching scheme by using the product as projection (incorporated in the branching rule **dsj**), apparently for the first time, while still not using look-ahead. [DABC96] (the **C-SAT** solver) did not yet use the product-projection, but introduced certain aspects of look-ahead (incorporating the first round of unit-clause propagation) and also a steeper decline for the weights w_k (close to the weights used by the **OKsolver-2002** scheme), namely now $w_k = -\log(1 - (2^{k+1} - 1)^{-2})$, where the quotient $\frac{w_k}{w_{k+1}}$ is monotonically decreasing, with limit 4. In Section 3.1 of [Ouy99] a possible derivation of this rule is offered. A similar approach (but already using the product-projection, and partially using r_2 as reduction), improving the **Satz** solver, is discussed in [Li99]. While these heuristics aimed at “doing a thorough job”, there is also the direction of “cheaper heuristics”, mostly focusing on efficient implementations. The basic idea is “MOM”, i.e., “maximum occurrences in clauses of minimum size”,

and all these heuristics can be understood as approximations of “maximise the number of new clauses”, driven by the kind of available data which happens to be supported by the data structures. The solver `Posit` ([Fre95]) introduced efficient data structures and an efficient implementation of r_2 -reduction (which altogether made it very competitive), but regarding heuristics no progress over [VT96] took place. Once the efficiency of r_2 -reduction (“failed literal eliminations”) for this type of solvers (the “look-ahead solvers”) became apparent, the cost of measurements needed to evaluate d_{MNC} and variations as well as the cost of look-aheads diminishes (relatively), and the branching scheme as embodied in its pure form by the `OKsolver-2002` (as discussed in Subsection 8.7.4.1) is now the common standard of look-ahead solvers. The `march`-solvers extended the scheme in various ways (see [Heu08]), considering only a subset of variables for the look-ahead, as pioneered by [LA97], but the core rule has been left unchanged.

8.8. Optimising distance functions

Now we turn to the topic of improving distance functions by evaluating how good they actually predict progress as given by the real branching trees. The possibility of evaluating distances leads to the possibility of actually optimising distances, and the two basic approaches are considered in Subsections 8.8.2, based on the evaluation techniques discussed in Subsection 8.8.1. The subject of optimising distances has been studied in the context of theoretical upper bounds (see Subsection 8.7.3). For practical SAT solving however there seems to be no general attempt (considering arbitrary distances) in the literature. The notion of “adaptive heuristics” as used in [Hv07] is based on a broad view of “heuristics”, and actually only concerns the reduction process (see Subsection 8.7.2), by dynamically adapting the set of variables eligible for reduction considerations, thus weakening the reduction by excluding variables, and also adapting the set of variables for a strengthened reduction.

8.8.1. Evaluating distance functions

The “relative” effectiveness of the distance function d used in a solver on one particular instance (with medium-size branching tree) can be gauged (that is, regarding its “relative values”) as follows: Compute the distribution of the random variable P^{-1} induced by \mathfrak{P}_d (see Definition 8.4.3) — a good distance function yields a centred distribution (around $E(P^{-1}) = \#lvs(T)$), while a bad distance function is more similar to the uniform distribution.

This evaluation is also computationally not very expensive: At each node along the current path the probabilities of the branches are computed, and once a leaf is found, then the product of these probabilities along the whole path is dumped to a file; viewing the distribution of the random variable P with a statistical tool kit (like R) might reveal interesting phenomena.⁸ That we consider only “relative values” of d here means that an “optimal distance” d (with constant

⁸Handling the logarithms of the probabilities is likely advantageous. And one should keep

$P^{-1} = \#\text{lvs}(T)$) is derived from a canonical distance $\Delta\mu$ with $\mu(v) = \log \#\text{lvs}(T_v)$ (recall Lemma 8.4.9) by multiplying the branching tuple at each inner node with an *individual* positive real number (possibly different for each node).

Care is needed with the interpretation of such evaluations: Comparing different distance functions can be misleading (even on the same tree), since, metaphorically speaking, to make some interesting predictions something needs to be risked, and a thumb heuristic can have a smaller variance than a more powerful heuristic.

A weakness in using the variance for evaluating distance functions is that the absolute values of the distances at different nodes are ignored. The second basic possibility for evaluating distances on (given) trees is to use the upper bound from Theorem 8.4.8, which on the other hand has the disadvantage that only the worst nodes are considered (while the variance includes all nodes).

8.8.2. Minimising the variance or minimising the tau-upper-bound

A numerical value for the quality of the distance function d is given by the variance $\text{Var}(P^{-1})$ (the smaller the better, and the optimal value is 0); for convenience the standard deviation $\sqrt{\text{Var}(P^{-1})}$ might be used, and for comparing different trees the normalised standard deviation $\frac{\sqrt{\text{Var}(P^{-1})}}{\#\text{lvs}(T)}$ is appropriate, however these quantities only serve as a “user interface” in our context, and are not considered furthermore. By Lemma 8.4.4 we have an efficient recursive method for computing the variance, which also does not cause space overhead since no additional storage is required (if we do not want to visually inspect the spread of P^{-1}). If we have given several distance functions d_1, \dots, d_k , then we can choose the best one (on average) as the one with minimal $\text{Var}(P_{d_i}^{-1})$. This method can also be used “on the fly”, during the execution of a backtracking solver, with negligible overhead, to dynamically adapt the solver to the problem at hand. Of course, measuring the quality of a distance function by the variance $\text{Var}(P_d^{-1})$ enables not only comparison of different distance functions, but if d depends on parameter α (possibly a vector), then we can also optimise d_α by minimising $\text{Var}(P_{d_\alpha}^{-1})$ (for the given tree). A special (interesting) case is where d is given as a convex linear combination of distances d_1, \dots, d_k (that is, $d = \sum_{i=1}^k \lambda_i \cdot d_i$ for $\lambda_i \geq 0$ and $\sum_{i=1}^k \lambda_i = 1$). Actually, here the d_i do not need to be distances, but could even assume negative values, if only we take care to restrict the λ_i accordingly to avoid non-positive values of d . These optimisations could be also performed “online”.

Instead of minimising $\text{Var}(P_{d_\alpha}^{-1})$ one could consider minimising $\max(P_{d_\alpha}^{-1}) = \max_{v \in \text{lvs}(T)} P_{d_\alpha}(v)^{-1}$, i.e., minimising the worst-case upper bound from Lemma 8.4.1. This task is considerably simplified by applying logarithms. However this optimisation is much rougher, since it only cares about getting rid off the most extreme cases (this is the worst-case perspective), which might be weak since the worst case might occur only rarely. So the only real alternative seems to minimise the upper bound $(\max \tau(d))^{\max \Sigma(d)}$, or better $\log((\max \tau(d))^{\max \Sigma(d)}) =$

in mind that on the leaves we do not consider the uniform probability distribution, and so just plotting the “observations “ is not meaningful.

$(\max \Sigma(d)) \cdot \log(\max \tau(d))$, which still suffers from considering only the worst case, but has the advantage that scaling of the distances is taken into account.

8.9. The order of branches

After having chosen the branching variable, the next step then is to order the branches, which according to Subsection 8.7.2 is the task of the *branching ordering heuristics* (thus ordering of branches is not integrated into the τ -method, but is an additional step).⁹ We extract an approximated “probability” that we have a favourable situation, like finding a satisfying assignment for look-ahead solvers, or “learning a lot” for conflict-driven solvers, and order the branches according to this approximation (in descending order). Unfortunately yet not much theoretically founded is known about clause-learning, and the question here is only how to compute “approximations” of some form of probability that a clause-set F is satisfiable. Such “approximations” are achieved by *satisfiability estimators* P .

For unsatisfiable problem instances the order doesn’t matter (without learning), while for satisfiable instances, whatever the branching is, finding the right first branch actually would solve the problem quickly (when ignoring the time for choosing the first branch). While in previous sections often the problem representation was not of importance, and very general satisfiability problems over non-boolean variables could be handled, now the approaches are combinatorially in nature and thus are more specific to (boolean) clause-sets. In Subsections 8.9.1 and 8.9.2 we consider the two schemes currently used in practice¹⁰. A difficulty is that these two satisfiability estimators for choosing the first branch do not have standard names; for ease of reference we propose the following names:

1. Since apparently the first approach has first been mentioned by John Franco, we call it the “Franco heuristics” or “Franco estimator”.
2. The second approach was apparently first mentioned in [Joh74] in the context of “straight-line programs”, which is close to our usage, while the “Jeroslow-Wang heuristics” from [JW90] misuses the heuristics to actually choose the branching *variable* (as discussed in Subsection 8.7.4.2), so it seems sensible to call the heuristics the “Johnson heuristics” (or the “Johnson estimator”).

8.9.1. The Franco estimator: Considering random clause-sets

This approach, apparently first mentioned in the literature in [GPFW97] (in the context of backtrack-free algorithms for random formulas), considers F as a

⁹In [Hv08] this is called “direction heuristics”. We prefer to speak of the ordering of branches since our methods work for arbitrarily wide branchings, and they consider not just the first branch.

¹⁰The *OKsolver-2002* uses the first scheme, and on random 3-CNF this scheme appears to be slightly stronger than the second one, an observation already made 1986 by John Franco in the context of (incomplete) “straight-line SAT algorithms” (which do not backtrack but abort in case of a failure).

random element $F \in \Omega(p_1(F), \dots, p_k(F))$ of some probability space Ω depending on parameters p_i . So now we can speak of the probability $P_\Omega(F \in \mathcal{SAT})$ that F is satisfiable. Yet this approach has not been applied in this form, and one does not consider the (complicated) probability $P_\Omega(F \in \mathcal{SAT})$ of “absolute” satisfiability, but instead the probability $P_\Omega(\varphi * F \in \mathcal{SAT})$ that a random (total) assignment φ satisfies a random $F \in \Omega$. Most natural is to consider the constant density model (with mixed densities), that is one considers all clause-sequences F as equally likely which have $n(F)$ many (formal) variables and $c(F)$ many clauses in total, where for clause-length k we have $c_k(F)$ many clauses. Then actually the random assignment φ_0 can be fixed, say to the all-zero assignment, and due to the independence of the clause-choices we have $P(\varphi_0 * F \in \mathcal{SAT}) = \prod_{C \in F} (1 - 2^{-|C|})$. For efficiency reasons the use of the logarithm $L(F) := \log \prod_{C \in F} (1 - 2^{-|C|}) = \sum_{C \in F} \log(1 - 2^{-|C|})$ is preferable, where furthermore the factors $\log(1 - 2^{-k})$ for relevant clause-lengths $k = 1, 2, 3, \dots$ can be precomputed. So the first approach yields the rule to order a given branching (F_1, \dots, F_m) by descending $L(F_i) = \sum_{k \in \mathbb{N}_0} c_k(F_i) \cdot \log(1 - 2^{-k})$.

8.9.2. The Johnson estimator: Considering random assignments

The second approach considers the specific F equipped with the probability space of all total assignments, and the task is to approximate the probability $\#\text{sat}(F)/2^{n(F)}$ that a random assignments satisfies F , where $\#\text{sat}(F)$ is the number of satisfying (total) assignment. Since for conjunctive normal forms falsifying assignments are easier to handle than satisfying assignments, we switch to the consideration of the probability $P_0(F) := \#\text{usat}(F)/2^{n(F)} = 1 - \#\text{sat}(F)/2^{n(F)}$ that a total assignment falsifies F (where $\#\text{usat}(F)$ is the number of total assignments falsifying F). We have the obvious upper bound $P_0(F) \leq \#\text{usat}(F)/2^{n(F)} \leq P_0^1(F) := \sum_{C \in F} 2^{-|C|}$, which is derived from considering the case that no total assignment falsifies two (different) clauses of F at the same time. The lower index “0” in $P_0^1(F)$ shall remind of “unsatisfiability”, while the upper index indicates that it is the first approximation given by the “inclusion-exclusion” scheme. Using P_0^1 as approximation to P_0 , we obtain the rule to order a branching (F_1, \dots, F_m) by ascending $\sum_{k \in \mathbb{N}_0} c_k(F_i) \cdot 2^{-k}$. We see that this method in principle is very similar to the first method, in both cases one minimises (for the first branch) the weighted number $c^w(F) = \sum_{k=0}^{\infty} w(k) \cdot c_k(F)$ of clauses of F , where the weights $w(k)$ depend only on the length k of the clauses. The only difference between these two methods are the weights chosen: for the first method we have $w_1(k) = -\log(1 - 2^{-k})$, for the second $w_2(k) = 2^{-k}$. Asymptotically we have $\lim_{k \rightarrow \infty} \frac{w_1(k)}{w_2(k)} = 1$, since for small x we have $-\log(1 - x) \approx x$. So the second method can also be understood as an approximation of the first method.

Another understanding of this heuristic comes from [Joh74]: $P_0^1(F)$ is the (exact) expected number of falsified clauses for a random assignment (this follows immediately by linearity of expectation). We remark that thus by Markov’s inequality we obtain again $P_0(F) \leq P_0^1(F)$. And finally we mention that if $P_0^1(F) < 1$ holds, then F is satisfiable, and the Johnson heuristic, used either without look-

ahead or with look-ahead involving unit-clause-propagation, will actually find a satisfying assignment without backtracking, since in case of $P_0^1(F) < 1$ for every variable $v \in \text{var}(F)$ there exists $\varepsilon \in \{0, 1\}$ with $P_0^1((v \rightarrow \varepsilon) * F) < 1$. We also see that using the Johnson heuristic without backtracking and without look-ahead yields an assignment which falsifies at most $P_0^1(F)$ many clauses of F . For fixed uniform clause-length k we get at most $P_0^1(F) = c(F) \cdot 2^{-k}$ falsified clauses, that is at least $c(F) \cdot (1 - 2^{-k})$ satisfied clauses, which actually has been shown in [Hås01] to be the optimal approximation factor ($\frac{1}{1-2^{-k}}$) for the maximal number of satisfied clauses which can be achieved in polynomial time (unless $P = NP$).

8.9.3. Alternative points of view

It appears to be reasonable that when comparing different branchings (for SAT this typically means different branching variables) where one branching has a branch with a very high associated probability of satisfiability, that then we take the satisfiability-aspect more important than the reduction-aspect, since we could be quite sure here. Yet it seems that due to the crudeness of the current schemes such considerations are not very successful with the methods discussed above, however they are applied in the context of a different paradigm, which does not use approximated satisfiability probabilities of problem instances for the ordering of branches, but uses approximations of *marginal probabilities* for single variables as follows: Consider a satisfiable problem instance F (for unsatisfiable instances the order of branches does not matter in our context) and a variable $v \in \text{var}(F)$. If we can compute reasonable approximations $\tilde{p}_v(\varepsilon)$ for values $\varepsilon \in \{0, 1\}$ of the ratio $p_v(\varepsilon)$ of satisfying (total) assignments f with $f(v) = \varepsilon$, then we choose first the branch ε with higher $\tilde{p}_v(\varepsilon)$. The main structural problem of this approach is that it focuses on single variables and cannot take further inferences into account, while when having a satisfiability probability estimator $P(F)$ at hand, then we can improve the accuracy of the approximation by not just considering $P(\langle v \rightarrow \varepsilon \rangle * F)$ but applying further inferences to $\langle v \rightarrow \varepsilon \rangle * F$. However especially for random problems this approach shows considerable success, and so we conclude this section by some pointers to the relevant literature.¹¹

The basic algorithm for computing the marginalised number of satisfying assignments (i.e., conditional on setting a variable to some given value) is the “sum-product algorithm” (see [KFL01]), also known as “belief propagation”. This algorithm is exact if the “factor graph”, better known to the SAT community as “clause-variable graph” (for clause-sets; a bipartite graph with variables and clauses as the two parts), is a tree. Considering the variable-interaction graph $\text{vig}(F)$ (nodes are variables, joined by an edge if occurring together in some constraint), in [KDG04] approximations $p_v^k(\varepsilon)$ of $p_v(\varepsilon)$ for a parameter $k \in \mathbb{N}_0$

¹¹Another problem with this approach is that for good predictions variables v with “strong bias”, i.e., with large $|\tilde{p}_v(0) - \tilde{p}_v(1)|$ are preferred, which interferes with the branching heuristics (for example on unsatisfiable instances preferring such variables seems rather senseless). This is somewhat similar to preferring a variable v with one especially high value $P(\langle v \rightarrow \varepsilon \rangle * F)$, but now the problem is even more pronounced, since a “confident” estimation $\tilde{p}_v(\varepsilon)$ requires $\tilde{p}_v(1 - \varepsilon)$ to be low, and such biased variables might not exist (even when using precise values).

are studied (for the purpose of value ordering) where the approximation is precise if $k \leq \text{tw}(\text{vig}(F))$, the treewidth of the variable-interaction graph. A recent enhancement of belief propagation is “survey propagation”; see [BMZ05] for the original algorithm, and [Man06, HM06] for further generalisations.

Finally, to conclude this section on the order of branches, some aspect is worth to mention which indicates that the order of branches should also take the (expected) complexity of the branches into account. Say we have a branching variable v , where branch $v \rightarrow 0$ has approximated SAT probability 0.7 and expected run time 1000s, while branch $r \rightarrow 1$ has approximated SAT probability 0.3 and expected run time 1s. Then obviously branch $v \rightarrow 1$ should be tried first. Apparently this approach has not been transformed yet into something really workable, likely for the same reason as with the “rational branching rule” mentioned in Subsection 8.2.1, namely that the estimations for running times are far too rough, but it explains the erratic success of methods for choosing the branching order according to ascending expected complexity (easiest problem first) in practice on selected benchmarks.

8.10. Beyond clause-sets

The general theory of branching heuristics, developed in Section 8.2 to Section 8.6, is applicable to any backtracking algorithm, including constraint satisfaction problems (with non-boolean variables), and also the methods discussed in Section 8.8 are applicable in general. However the special heuristics discussed in Section 8.7 and Section 8.9 focused on SAT for boolean CNF. In this section we give an outlook on heuristics using more general “constraints” than clauses, considering “generalised clauses” in Subsection 8.10.1 (for examples BDD’s), which support generic SAT solving (via the application of partial assignments), and considering representations from constraint programming in Subsection 8.10.2.

8.10.1. Stronger inference due to more powerful “clauses”

Staying with boolean variables, a natural candidate for strengthened clauses are OBDDs. [FKS⁺04] is a good example for this direction, which also contains an overview on other approaches. Actually, OBDDs are only a stepping stone for [FKS⁺04], and the form of of generalised clauses actually used are “OBDDs on steroids”, called “smurf’s”, which store for every partial assignment (in the respective scope) all inferred assignments. In accordance with the general scheme MNC from Subsection 8.7.4.1, but without (i.e., with trivial) look-ahead¹², [FKS⁺04] define for each smurf a weight after the branching assignment, which reflects the reduction in the number of variables of the smurf due to inferred assignments, directly and with exponential decay also for the possible futures. More lightweight approaches include equivalence reasoning; see [Li03] for heuristical approaches to include the strengthened reasoning efficiently into the look-ahead of the heuristics.

¹²likely due to the fact that standard clause-learning is used, whose standard data structures are incompatible with stronger reductions and look-aheads

8.10.2. Branching schemes for constraint satisfaction

While the previous Subsection 8.10.1 considered “generalised SAT” in the sense that, though more general “constraints” are used, they are very “structured” w.r.t. allowing efficient inspection of the effects of (arbitrary) partial assignments. The field of constraint satisfaction on the other side tends to take a black-box point of view of constraints. The theory developed in this chapter yields a straightforward canonical basic heuristics for this environment, and we present these considerations below. First however some general comments on the discussion of branching heuristics for constraint satisfaction problems (CSPs) in [van06].

The *branching strategy* (see Section 4.2 in [van06]) selects the way in which to split the problem. For SAT solving binary branching on a variable is absolutely dominant (only in theoretical investigations more complex branchings are used), and thus for practical SAT solving the problem of the choice of branching strategy does not exist (at this time): An essential feature of SAT solving (compared for example with CSP solving) is that problems are “shredded” into tiny pieces so that the “global intelligence” of a SAT solver can be applied, and only “micro decisions” are made (via *boolean* variables), always on the outlook for a better opportunity (which might arise later, while by a more complex branching we might have made choices too early). Translating a problem with non-boolean variables to SAT (with boolean variables) typically increases the search space. On the other hand, [MH05] showed that this increased search space also contains better branching possibilities: A d -way branching for a variable v with domain D_v of size d (i.e., $v = \varepsilon_1, v = \varepsilon_2, \dots, v = \varepsilon_d$ for $D_v = \{\varepsilon_1, \dots, \varepsilon_d\}$) can always be efficiently simulated by a 2-way branching (corresponding to $v = \varepsilon, v \neq \varepsilon$ for some $\varepsilon \in D_v$), but not vice versa.

The *variable ordering heuristics* (see Section 4.6.1 in [van06]) is responsible for choosing the branching variable (given that only branching on a single variable is considered), which for SAT solving is typically just called the “branching heuristics”. The general tendency seems to be also to choose a variable minimising the expected workload, but surprisingly the integration of the information on the single branches (for the different values in the domain of the variable) into one single number, i.e., the task of the evaluation projection, has apparently never been systematically considered, and consequently the handling of projection is weak. Thus, though 2-way branching can have an edge, as mentioned above, there seems to be some scope of improvement for existing d -way branching (and other schemes yielding non-binary branching) by using the τ -projection (which plays out its strength in a context where various branching widths occur!).

The *value ordering heuristics* (see Section 4.6.2 in [van06]) is responsible for choosing the order of the branches (compare Section 8.9 in this chapter). In (standard, i.e., binary) SAT branching this just means the choice of the first branch.

Now to the most basic measures and distances. The basic estimation for problem complexity for boolean problems F is $2^{n(F)}$, the size of the search tree, from which we derive the measure $\mu(F) = n(F) = \log_2 2^{n(F)}$ (motivated by Lemma

8.4.9). As already mentioned at several places, using the distance Δn is rather weak for general SAT solving. However this measure can be strengthened by either strengthening the look-ahead, or by differentiating more between variables. The latter is possible for CSP problems, where variables v have domains D_v , and then $\mu(F) = \log \prod_{v \in \text{var}(F)} |D_v| = \sum_{v \in \text{var}(F)} \log(|D_v|)$ becomes a “more informed” measure of problem complexity (note that $\mu(F) = n(F)$ if all variables are binary). When considering arbitrary branching width, usage of the τ -function becomes compulsory, and comparing different branchings $F \rightsquigarrow (F_1^i, \dots, F_{p_i}^i)$ is then done by choosing the branching i with minimal $\tau(\Delta\mu(F, F_1^i), \dots, \Delta\mu(F, F_{p_i}^i))$. When combined with look-ahead (recall, this always refers to the branching heuristics (alone), that is, the F_j^i are better approximations of the reduced formula computed when actually choosing this branch), this yields decent basic performance. However if more information on the constraints is available, then the approach as discussed at the end of Subsection 8.7.4.1 likely will yield stronger results.

8.11. Conclusion and outlook

Regarding branching heuristics (the main topic of this chapter), one could say that historically the first phase has been finished by now, laying the foundations by constructing basic heuristics, as outlined in Subsection 8.7.4, and developing the basic theory as outlined in Sections 8.3 to 8.5. Now a much stronger emphasise should be put on precise quantitative analysis, as discussed in Section 8.8 on optimising (and measuring) the quality of heuristics, including the consideration of specialised branching projections as discussed in Section 8.6. Case studies on classes of problems should be valuable here. When saying that for branching heuristics the initial phase of theory building and systematic experimentation is finished, then actually this can only be said about finding the branchings, while regarding ordering the branches (as surveyed in Section 8.9) we are lacking a good theoretical foundation, and likely also much further experimentation is needed.

The general theory (on finding good branchings and ordering them) is applicable also beyond clause-sets (as discussed in Section 8.10), and while for clause-sets we have at least reasonable intuitions what might be good measures (distances) and ordering heuristics, here the situation is much more complicated. Combining the theory developed in this chapter with the different approaches from the field of constraint satisfaction (which in my opinion lacks the general theory, but has a lot to say on specific classes of constraints) should be a good starting point.

Finally, a major enterprise lies still ahead of us: theoretical foundations for heuristics for conflict-driven SAT solvers. In the introductions to Sections 8.7 and 8.9 we made a few remarks on conflict-driven solvers, but mostly we did not cover them in this chapter due to their non-tree-based approach and the lack of a theoretical foundation. See Chapter 4 of this handbook for more information on these solvers and the underlying ideas, where the basic references are as follows: With the solver **GRASP** ([MSS99]) the basic ideas got rolling, then **Chaff** ([MMZ⁺01, ZMMM01]) was considered by many as a breakthrough, while further progress was obtained by the solvers **BerkMin** ([GN02]) and **MiniSat** ([ES04]).

References

- [BF81] Avron Barr and Edward A. Feigenbaum, editors. *The Handbook of Artificial Intelligence, Volume I*. William Kaufmann, Inc., 1981. ISBN 0-86576-005-5.
- [BKD⁺04] Per Bjesse, James Kukula, Robert Damiano, Ted Stanion, and Yunshan Zhu. Guiding SAT diagnosis with tree decompositions. In Giunchiglia and Tacchella [GT04], pages 315–329. ISBN 3-540-20851-8.
- [BKS04] Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004.
- [BMZ05] A. Braunstein, M. Mézard, and R. Zecchina. Survey propagation: An algorithm for satisfiability. *Random Structures and Algorithms*, 27(2):201–226, March 2005.
- [Bul03] P. S. Bullen. *Handbook of Means and Their Inequalities*, volume 560 of *Mathematics and Its Applications*. Kluwer Academic Publishers, 2003. ISBN 1-4020-1522-4.
- [CFR05] Paul Cull, Mary Flahive, and Robby Robson. *Difference Equations: From Rabbits to Chaos*. Undergraduate Texts in Mathematics. Springer, 2005. ISBN 0-387-23234-6.
- [DABC96] Olivier Dubois, P. Andre, Y. Boufkhad, and C. Carlier. SAT versus UNSAT. In Johnson and Trick [JT96], pages 415–436. The Second DIMACS Challenge.
- [Epp06] David Eppstein. Quasiconvex analysis of multivariate recurrence equations for backtracking algorithms. *ACM Transactions on Algorithms*, 2(4):492–509, October 2006.
- [ES04] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Giunchiglia and Tacchella [GT04], pages 502–518. ISBN 3-540-20851-8.
- [FGK05] Fedor V. Fomin, Fabrizio Grandoni, and Dieter Kratsch. Some new techniques in design and analysis of exact (exponential) algorithms. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 87:47–77, October 2005.
- [FKS⁺04] John Franco, Michal Kouril, John Schlipf, Sean Weaver, Michael Dransfield, and W. Mark Vanfleet. Function-complete lookahead in support of efficient SAT search heuristics. *Journal of Universal Computer Science*, 10(12):1655–1692, 2004.
- [Fre95] Jon William Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, University of Pennsylvania, 1995.
- [GN02] Evgenii I. Goldberg and Yakov Novikov. Berkmin: A fast and robust Sat-solver. In *DATE*, pages 142–149. IEEE Computer Society, 2002.
- [GPFW97] Jun Gu, Paul W. Purdom, John Franco, and Benjamin W. Wah. Algorithms for the satisfiability (SAT) problem: A survey. In Dingzhu Du, Jun Gu, and Panos M. Pardalos, editors, *Satisfiability Problem:*

- Theory and Applications (DIMACS Workshop March 11-13, 1996)*, volume 35 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 19–151. American Mathematical Society, 1997. ISBN 0-8218-0479-0.
- [GT04] Enrico Giunchiglia and Armando Tacchella, editors. *Theory and Applications of Satisfiability Testing 2003*, volume 2919 of *Lecture Notes in Computer Science*, Berlin, 2004. Springer. ISBN 3-540-20851-8.
- [Hås01] Johan Håstad. Some optimal inapproximability results. *Journal of the ACM*, 48(4):798–859, July 2001.
- [Heu08] Marijn J. H. Heule. *SMART solving: Tools and techniques for satisfiability solvers*. PhD thesis, Technische Universiteit Delft, 2008. ISBN 978-90-9022877-8.
- [HLP99] G. H. Hardy, J. E. Littlewood, and G. Pólya. *Inequalities*. Cambridge Mathematical Library. Cambridge University Press, second edition, 1999. ISBN 0-521-35880-9; reprint of the second edition 1952.
- [HM06] Eric I. Hsu and Sheila A. McIlraith. Characterizing propagation methods for boolean satisfiability. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006*, volume 4121 of *Lecture Notes in Computer Science*, pages 325–338. Springer, 2006. ISBN 3-540-37206-7.
- [HUL04] Jean-Baptiste Hiriart-Urruty and Claude Lemaréchal. *Fundamentals of Convex Analysis*. Grundlehren (text editions). Springer, 2004. ISBN 3-540-42205-6; second corrected printing.
- [HV95] John N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15:359–383, 1995.
- [Hv07] Marijn J. H. Heule and Hans van Maaren. Effective incorporation of double look-ahead procedures. In Joao P. Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing - SAT 2007*, volume 4501 of *Lecture Notes in Computer Science*, pages 258–271. Springer, 2007. ISBN 978-3-540-72787-3.
- [Hv08] Marijn J. H. Heule and Hans van Maaren. Whose side are you on? Finding solutions in a biased search-tree. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:117–148, 2008.
- [Joh74] David S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9(3):256–278, December 1974.
- [JT96] David S. Johnson and Michael A. Trick, editors. *Cliques, Coloring, and Satisfiability*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996. The Second DIMACS Challenge.
- [JW90] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.
- [KDG04] Kalev Kask, Rina Dechter, and Vibhav Gogate. Counting-based look-ahead schemes for constraint satisfaction. In *Principles and Practice*

- of *Constraint Programming (CP 2004)*, volume 3258 of *Lecture Notes in Computer Science (LNCS)*, pages 317–331, 2004.
- [KFL01] Frank R. Kschischang, Brendan J. Frey, and Hans-Andrea Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47(2):498–519, February 2001.
- [KL97] Oliver Kullmann and Horst Luckhardt. Deciding propositional tautologies: Algorithms and their complexity. Preprint, 82 pages; the ps-file can be obtained at <http://cs.swan.ac.uk/~csoliver/>, January 1997.
- [KL98] Oliver Kullmann and Horst Luckhardt. Algorithms for SAT/TAUT decision based on various measures. Preprint, 71 pages; the ps-file can be obtained from <http://cs.swan.ac.uk/~csoliver/>, December 1998.
- [Knu75] Donald E. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29(129):121–136, January 1975.
- [Kul92] Oliver Kullmann. Obere und untere Schranken für die Komplexität von aussagenlogischen Resolutionsbeweisen und Klassen von SAT-Algorithmen. Master's thesis, Johann Wolfgang Goethe-Universität Frankfurt am Main, April 1992. (Upper and lower bounds for the complexity of propositional resolution proofs and classes of SAT algorithms (in German); Diplomarbeit am Fachbereich Mathematik).
- [Kul98] Oliver Kullmann. Heuristics for SAT algorithms: A systematic study. In *SAT'98*, March 1998. Extended abstract for the Second workshop on the satisfiability problem, May 10 - 14, 1998, Eringerfeld, Germany.
- [Kul99a] Oliver Kullmann. Investigating a general hierarchy of polynomially decidable classes of CNF's based on short tree-like resolution proofs. Technical Report TR99-041, Electronic Colloquium on Computational Complexity (ECCC), October 1999.
- [Kul99b] Oliver Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, 223(1-2):1–72, July 1999.
- [Kul02] Oliver Kullmann. Investigating the behaviour of a SAT solver on random formulas. Technical Report CSR 23-2002, Swansea University, Computer Science Report Series (available from <http://www-compsci.swan.ac.uk/reports/2002.html>), October 2002.
- [Kul03] Oliver Kullmann. Lean clause-sets: Generalizations of minimally unsatisfiable clause-sets. *Discrete Applied Mathematics*, 130:209–249, 2003.
- [Kul08a] Oliver Kullmann. Fundamentals of branching heuristics: Theory and examples. Technical Report CSR 7-2008, Swansea University, Computer Science Report Series (<http://www.swan.ac.uk/compsci/research/reports/2008/>), April 2008.
- [Kul08b] Oliver Kullmann. Present and future of practical SAT solving. In Nadia Creignou, Phokion Kolaitis, and Heribert Vollmer, editors, *Complexity of Constraints*, volume 5250 of *Lecture Notes in Com-*

- puter Science (LNCS)*, pages 283–319. Springer, 2008.
- [LA97] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 366–371. Morgan Kaufmann Publishers, 1997.
- [Li99] Chu Min Li. A constraint-based approach to narrow search trees for satisfiability. *Information Processing Letters*, 71(2):75–80, 1999.
- [Li03] Chu Min Li. Equivalent literal propagation in the DLL procedure. *Discrete Applied Mathematics*, 130:251–276, 2003.
- [Luc84] Horst Luckhardt. Obere Komplexitätsschranken für TAUT-Entscheidungen. In *Frege Conference 1984, Schwerin*, pages 331–337. Akademie-Verlag Berlin, 1984.
- [Man06] Elitza Nikolaeva Maneva. *Belief propagation algorithms for constraint satisfaction problems*. PhD thesis, University of California, Berkeley, 2006.
- [MH05] David G. Mitchell and Joey Hwang. 2-way vs. d-way branching for CSP. In Peter van Beek, editor, *Principles and Practice of Constraint Programming — CP 2005*, volume 3709 of *Lecture Notes in Computer Science*, pages 343–357. Springer, 2005.
- [MMZ⁺01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *DAC*, pages 530–535. ACM, 2001.
- [MS85] B. Monien and Ewald Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics*, 10:287–295, 1985.
- [MSS99] Joao P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.
- [Ouy99] Ming Ouyang. *Implementations of the DPLL algorithm*. PhD thesis, Graduate School—New Brunswick; Rutgers, The State University of New Jersey, 1999.
- [Ren92] J. Renegar. On the computational complexity and geometry of the first-order theory of the reals. *Journal of Symbolic Computation*, 13:255–352, 1992.
- [Sch92] Ingo Schiermeyer. Solving 3-satisfiability in less than 1.579^n steps. In *Selected papers from Computer Science Logic '92*, volume 702 of *Lecture Notes Computer Science*, pages 379–394, 1992.
- [van06] Peter van Beek. Backtracking search algorithms. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, Foundations of Artificial Intelligence, chapter 4, pages 85–134. Elsevier, 2006. ISBN 0-444-52726-5.
- [VT96] Allen Van Gelder and Yumi K. Tsuji. Satisfiability testing with more reasoning and less guessing. In Johnson and Trick [JT96], pages 559–586. The Second DIMACS Challenge.
- [vW00] Hans van Maaren and Joost Warners. Solving satisfiability problems using elliptic approximations — effective branching rules. *Discrete*

Applied Mathematics, 107:241–259, 2000.

- [Wah07] Magnus Wahlström. *Algorithms, Measures and Upper Bounds for Satisfiability and Related Problems*. PhD thesis, Linköpings universitet, Department of Computer and Information Science, SE-581 83 Linköping, Sweden, 2007. ISBN 978-91-85715-55-8.
- [ZMMM01] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pages 279–285. IEEE Press, 2001.

Chapter 9

Preprocessing in SAT Solving

Armin Biere, Matti Järvisalo, and Benjamin Kiesl

Preprocessing has become a key component of the Boolean satisfiability (SAT) solving workflow. In practice, preprocessing is situated between the encoding phase and the solving phase, with the aim of decreasing the total solving time by applying efficient simplification techniques on SAT instances to speed up the search subsequently performed by a SAT solver. In this chapter, we overview key preprocessing techniques proposed in the literature. While the main focus is on techniques applicable to formulas in conjunctive normal form (CNF), we also selectively cover main ideas for preprocessing structural and higher-level SAT instance representations.

9.1. Introduction

One of the main reasons for the success of SAT solving is the ability of modern conflict-driven clause-learning solvers (described in Chapter 4 on CDCL) to search over large CNF formulas with millions of variables and clauses. Nevertheless, real-world applications such as model checking (Chapter 18) and software verification (Chapter 20) often yield formulas that are still too large for state-of-the-art solvers, which in turn can limit the applicability of SAT solving as the solution method of choice. The immense size of formulas is often caused by a combination of the complex nature of the problems themselves, the propositional encodings available, and the automated encoding techniques that generate the formulas in practice.

Automated encoding techniques in particular can deteriorate solver performance by introducing redundancies in the form of unnecessary clauses and variables. Moreover, one and the same problem can often be encoded in various different ways, meaning that SAT solvers are at the mercy of the user in terms of how a problem instance is actually encoded. This calls for *preprocessing*, that is, the application of automated techniques that simplify formulas before passing them to solvers in order to improve the efficiency of the whole solving pipeline. Preprocessing not only improves efficiency in general, it also takes the burden of crafting complex problem encodings from the user, thus allowing non-expert users to leverage the power of modern SAT solvers.

Preprocessing has within recent years become a key component of the SAT solving workflow. A landmark in the history of preprocessing is the technique of *bounded variable elimination* (BVE) [EB05], as implemented in the SATELITE preprocessor [EB05] and later integrated in version 2.0 [ES06] of MINISAT [ES03].¹ In 2005 and 2006 it contributed to the largest improvement of solver performance witnessed in the history of the SAT competitions. Even today, bounded variable elimination is still arguably the most important practical preprocessing technique.

In the SAT solving workflow, preprocessing is situated between the encoding phase, during which a user or a dedicated tool encodes a problem as a formula of propositional logic, and the solving phase, during which a SAT solver tries to determine the satisfiability of a formula. The distinction between these phases, however, is not straightforward: on the one hand, preprocessing can be viewed as an automated re-encoding of a formula, which would make it part of the encoding phase; on the other hand, it can also be viewed as a form of reasoning itself, which would make it part of the solving phase. The latter view is emphasized by the recent rise of so-called *inprocessing* SAT solvers [JHB12, FBS19], which interleave search with phases during which typical preprocessing techniques are applied. In fact, since 2013 the SAT competitions have been dominated by CDCL solvers that utilize inprocessing. However, combining incremental SAT solving [Hoo93, KWS00, WKS01, ES03, ALS13, BBIS16] with preprocessing and inprocessing still poses a challenge in practice [KNPH05, KN11, NRS12, NR12, NRS14, FBS19] and will not be considered much further in this chapter.

From an abstract point of view, preprocessing techniques can be seen as implementations of inference rules that, when applied to formulas arising from real-world problem domains, yield equisatisfiable formulas that are easier to solve. Most preprocessing techniques are designed to simplify CNF formulas on the syntactic level by reducing the number of variables and the number of clauses. A popular example of this is again the method of bounded variable elimination, which uses Davis-Putnam resolution [DP60] and other techniques to eliminate variables without increasing the number of clauses.

Reducing the size of formulas, however, does not always improve the efficiency of the solving process. Sometimes SAT solvers actually benefit from redundant formula parts, which makes the search for effective preprocessing more complex than the search for methods that just yield equisatisfiable formulas of smaller size. The most evident example of this is the clause learning mechanism at the heart of CDCL solvers (Chapter 4 on CDCL): clause learning drastically improves the solving process by adding clauses that are logically entailed by the input formula and thus, by definition, redundant. Additionally, binary clauses (i.e., clauses with only two literals) are often considered special and thus not removed by SAT solvers due to their high potential for enabling more propagation.

A drastic example illustrating the worst-case impact of preprocessing techniques that *remove* clauses comes from the theoretical study of proof systems (Chapter 7): In 1985, Haken proved that a natural SAT encoding of the pigeon-hole principle (PHP) admits only resolution proofs of exponential size [Hak85], meaning that CDCL solvers—which are based on resolution—require exponential

¹While tight integration of preprocessing with search is now more common there also exist stand-alone preprocessing tools [Man12, WvdGS13] in the spirit of SATELITE [EB05].

time to solve this encoding [BKS04, PD11]. But, as shown by Cook [Coo76], if we add specific clauses (so-called *definition clauses*) to the PHP encoding, the resulting CNF formulas have polynomial-size resolution proofs, meaning that CDCL solvers can, at least in theory, solve them in polynomial time.

Unfortunately, popular preprocessing techniques such as *bounded variable elimination*, *blocked clause elimination* [JBH10], and *cone-of-influence reduction* (a circuit-preprocessing technique described in Chapter 27) remove these definition clauses, turning an instance that is tractable for CDCL solvers into an intractable one. In practice, some of the most effective preprocessing techniques still aim at removing formula parts, but others also add strong redundant clauses such as binary or even unit clauses.

Another approach to analyze the effects and limits of preprocessing is offered by *kernelization* as studied in the area of fixed-parameter complexity, the topic of Chapter 17. However, while kernelization is an important theoretical concept in the context of SAT, it has—unlike the other techniques discussed in this chapter—so far not been used effectively in practical preprocessing.

Classifying practical preprocessing techniques in terms of their underlying inference rules is difficult, but we can identify two main categories: The first category consists of techniques that employ the resolution rule and its variations; examples are *unit propagation*, *bounded variable elimination* [SP04, Bie04, EB05], *hyper binary resolution* [Bac02, BW03, GS05, HJB13], and other advanced probing techniques such as *distillation* [HS07] and *vivification* [PHS08].

The second category consists of preprocessing techniques that identify literals or clauses that are redundant under different notions of redundancy; examples of such techniques are *clause subsumption*, *hidden literal elimination* [HJB11], *equivalent-literal substitution* (called *equivalence reduction* in [Bac02, BW03]), and *blocked clause elimination* [JBH10] together with its generalizations [HJL⁺15, KSTB16, KSTB18, KSS⁺17, HKB17, HKB20].

Preprocessing techniques also vary in the ways in which they analyze formulas. Most techniques are applied syntactically on CNF formulas, but some utilize graph representations of implication chains and other types of functional dependencies, or even higher-level constraints—implicitly represented by clauses—to preprocess a formula.

Although many individual preprocessing techniques are by themselves quite simple, the potential of preprocessing and inprocessing lies in combining different preprocessing techniques in such a way that they benefit from each other. For instance, at points where one technique is unable to make further progress, another technique might be applicable, and might even modify the formula in ways that trigger the first technique again.

In practice, however, running a specific preprocessing technique until completion is often only feasible for techniques that are computationally inexpensive. Because of this, even stand-alone preprocessors such as SATELITE [EB05] rely on limits that bound the maximum number of occurrences of a candidate variable to be eliminated. Without such limits they would not be beneficial and run out of time on many instances. Moreover, scheduling the effort spent on individual preprocessing techniques is not trivial and has not seen much published research to date [Bie14]. Different preprocessing techniques can also have similar effects

in the sense that they perform the same modifications; for instance, both blocked clause elimination and bounded variable elimination have been shown to simulate cone-of-influence reduction on the CNF-level [JBH12].

Finally, the naive incorporation of preprocessing and inprocessing into the solving pipeline can potentially lead to unexpected outcomes, including incorrect solving results. Especially in inprocessing SAT solvers, a lot of care [JHB12] is required to ensure that simplification techniques interact in a sound way with both clause learning and *clause forgetting* (learned clauses are *deleted* quite frequently during the *reduce* phase of a SAT solver to improve memory usage and speed up propagation). Additionally, since many preprocessing techniques only maintain equisatisfiability but not logical equivalence, solution-reconstruction methods are sometimes required to transform a satisfying assignment of a preprocessed formula into a satisfying assignment of the original formula.

Most techniques described in this chapter are also available in the SAT solver CADICAL [Bie17]. The goal of developing CADICAL was to produce a SAT solver that is thoroughly documented, easy to understand and modify, and in the end still comparable to the state of the art (in the SAT Race 2019 it actually solved the largest number of instances). Therefore, consulting the CADICAL source code and its (inlined) documentation at <https://github.com/arminbiere/cadical> in parallel to reading this chapter is highly recommended. As CADICAL has been participating in the SAT competition since 2017, its solver descriptions [Bie17, Bie18, Bie19, BFFH20] provide additional technical details.

In the rest of this chapter, we give an overview of key preprocessing techniques proposed in the literature. Although our main focus is on CNF-level preprocessing, we will also briefly discuss main ideas proposed for structure-based preprocessing on the level of Boolean circuit representations of propositional formulas. The chapter is organized into sections describing CNF-level techniques, from basic concepts such as unit propagation (Section 9.2), over more advanced resolution-based preprocessing such as bounded variable elimination (Section 9.3), to preprocessing beyond resolution-based techniques, such as clause elimination (Section 9.4), as well as how to reconstruct solutions to original input CNF formulas after preprocessing (Section 9.5). Beyond CNF-level techniques we provide an additional shorter overview of preprocessing techniques applicable on higher-level propositional representations (Section 9.6), and conclude the chapter with a short summary and future work (Section 9.7).

9.2. Classical Preprocessing Techniques

We begin by discussing various classical CNF-level preprocessing techniques: unit propagation (the main propagation mechanism in complete SAT solvers), pure literal elimination, basic clause elimination techniques, and the detection of connected components in a graph representation of formulas to identify formula parts that can be solved independently of each other.

9.2.1. Unit Propagation

Unit propagation is a straightforward technique that has been common in SAT solving for many decades [DP60]; it is based on the *unit-clause rule*: Given a

formula in conjunctive normal form, the unit-clause rule checks if the formula contains a *unit clause*, i.e., a clause that contains only a single literal. If so, it removes all clauses that contain the literal from the formula (this step can be seen as an instance of subsumed-clause removal) and it removes the negation of the literal from the other clauses. Unit propagation repeatedly applies the unit-clause rule until either it derives the empty clause or there are no more unit clauses left. In the former case, we say that unit propagation derives a *conflict*.

Example 1. Consider the formula $(x) \wedge (\bar{x} \vee y) \wedge (\bar{y} \vee z \vee v)$. The unit-clause rule takes the unit clause (x) and first removes it from the formula. It then removes the literal \bar{x} from the clause $(\bar{x} \vee y)$, resulting in the formula $(y) \wedge (\bar{y} \vee z \vee v)$. Another application of the unit-clause rule first removes the unit clause (y) and then removes the literal \bar{y} from $(\bar{y} \vee z \vee v)$. We end up with the formula $(z \vee v)$.

Clearly, whenever a formula contains a unit clause, the formula can only be satisfied by assignments that make the literal of the unit clause true. The application of unit propagation during preprocessing is therefore safe insofar as the resulting formula and the original formula are equisatisfiable. If unit propagation derives a conflict, we can conclude that the formula must be unsatisfiable. The converse, however, is not true. For example, the formula $(x \vee y) \wedge (\bar{x} \vee y) \wedge (x \vee \bar{y}) \wedge (\bar{x} \vee \bar{y})$ is unsatisfiable, but since it does not contain any unit clauses, unit propagation cannot derive a conflict.

Unit propagation (as well as several other preprocessing techniques we discuss later on) can be seen as the repeated application of the classical *resolution* rule [Rob65]. Given two clauses $(c_1 \vee \dots \vee c_m \vee l)$ and $(d_1 \vee \dots \vee d_n \vee \bar{l})$, the resolution rule allows us to derive the clause $(c_1 \vee \dots \vee c_m \vee d_1 \vee \dots \vee d_n)$, which is called the *resolvent* of the two clauses upon l . In case of unit propagation, the removal of the literal \bar{l} from a clause can be seen as a resolution step with the clause (l) and thus is also often called *unit resolution*. In the context of constraint programming, unit propagation is also called *Boolean constraint propagation* (BCP). Unit propagation is essential for the original search-based D(P)LL algorithm [DLL62] as well as for the CDCL procedure described in Chapter 4.

9.2.2. Failed Literals

The notion of a *failed literal* [LB01, LaPMS03] forms the basis of a preprocessing technique known as *failed-literal probing*, which is an integral part of look-ahead solving (see Chapter 5). The origin of failed literals dates back to Zabih and McAllester [ZM88], and is featured in the thesis of Freeman [Fre95]. The related SLUR algorithm by Schlipf et. al. [SAFS95] has been used to characterize a class of polynomially solvable formulas (see also [dV00] for a related line of work).

A literal l is a failed literal with respect to a formula F if unit propagation derives a conflict on $F \wedge (l)$. Clearly, if unit propagation derives a conflict on $F \wedge (l)$, then $F \wedge (l)$ is unsatisfiable and therefore F implies \bar{l} . The addition of the unit clause (\bar{l}) to F thus preserves equivalence, and the corresponding preprocessing technique that adds the negations of failed literals to a formula is called failed-literal probing.

Example 2. Let $F = (x \vee u) \wedge (\bar{x} \vee u) \wedge (\bar{u} \vee z \vee y) \wedge (\bar{u} \vee z \vee \bar{y})$. Since F does not contain any unit clauses, it cannot be simplified by unit propagation alone. However, \bar{u} is a failed literal because unit propagation derives a conflict on $F \wedge (\bar{u})$, caused by the first two clauses. The unit clause (u) is therefore implied by F and so we can add it to F . If we then perform unit propagation on $F \wedge (u)$, we obtain the simplified formula $G = (z \vee y) \wedge (z \vee \bar{y})$.

In practical preprocessing, a list of candidate literals is considered, and for all the candidate literals it is checked whether or not they are failed literals. It is generally not clear how to order the candidate literals to minimize the number of required decisions. For instance, in Example 2, the literal \bar{u} is the only failed literal with respect to F . In the worst case, we would thus have tested all other literals before (\bar{u}) . After simplifying F , which yields G , we can see that \bar{z} becomes a failed literal, but z , y , and \bar{y} are still no failed literals. Simplifying a formula until there are no more failed literals can thus require us to test quadratically many literals (with respect to the number of variables) in the worst case. Assuming the strong exponential-time hypothesis (ETH), the quadratic upper bound is tight in terms of both the number of variables and the number of clauses, even for Horn-3-CNF formulas [JK14]. As each test performs unit propagation, this results in cubic accumulated complexity in the worst case.

The cubic complexity can be reduced by marking the literals that were implied by unit propagation in tests since the last time a failed literal has been detected. These literals do not have to be tested until another failed literal is found, since their propagation did not lead to a conflict. As soon as a test yields a conflict—meaning that a new failed literal is found—all the marked literals have to be unmarked again (it is more efficient to use time-stamping techniques though). This optimization was independently discovered and described in [ABS99] and [SNS02].

Another optimization consists of testing only literals that do not occur in binary clauses themselves (but their negation does), thus effectively only testing the roots of the binary implication graph [GS05, Bie17].² With these techniques, failed-literal probing can run until completion even on large industrial SAT instances or can be used as preprocessing for splitting and partitioning in distributed SAT solving [HJN10]. Later in this chapter we will discuss an extension, based on look-ahead (see also Chapter 5), that learns binary clauses on the fly by combining *hyper binary resolution* with equivalent-literal reasoning.

9.2.3. Pure Literals

A pure literal is a literal whose negation does not occur in the formula. The corresponding technique of pure-literal elimination removes all clauses that contain a pure literal, because these clauses can be trivially satisfied by making the pure literal true without falsifying any other clauses.

Example 3. Consider the formula $(\bar{x} \vee y \vee z) \wedge (\bar{y} \vee \bar{z})$. The literal \bar{x} is a pure literal in this formula whereas all the other literals are not pure.

²When focusing on roots, cyclic literal dependencies, as in $(a \vee \bar{b}) \wedge (\bar{a} \vee b) \wedge (a \vee b)$, might prevent detection of some failed literals though. This can be avoided by computing strongly connected components and, for instance, equivalent-literal substitution as discussed in Section 9.3.2.

Pure-literal elimination was introduced independently both by Davis and Putnam [DP60] and by Dunham, Fridshal, and Sward [DFS59]. While a failed literal can be seen as a literal which we make false because making it true would lead to a conflict, a pure literal can be seen as a literal which we make true because making it true cannot lead to a conflict. Moreover, if a formula is unsatisfiable, clauses containing a pure literal cannot be used productively in a resolution proof because pure literals can never be resolved away.

An alternative to removing clauses that contain pure literals is to add the pure literals as unit clauses. The addition of these unit clauses, however, does not always preserve logical equivalence, because it could remove potential models in which a pure literal is false. Nevertheless, adding pure literals as unit clauses does not change the satisfiability status of a formula. This idea of performing transformations that yield equisatisfiable—but not necessarily logically equivalent—formulas is the corner stone of more advanced preprocessing techniques, such as blocked clause addition, which we discuss later.

9.2.4. Subsumption

A clause C is *subsumed* by a clause D if the set of literals in C is a superset of the literals in D . Analogously, a clause is subsumed by a formula if it is subsumed by some clause in the formula. The removal of subsumed clauses is an important technique that is often referred to as *subsumption*. The idea behind subsumption is to eliminate clauses that are redundant in the sense that their removal does not affect the satisfiability of a formula: if a formula contains two clauses C and D , where D subsumes C , the removal of C yields a logically equivalent formula since every assignment that satisfies D must also satisfy C .

Example 4. The clause $(\bar{x} \vee y \vee z)$ is subsumed by the clause $(\bar{x} \vee z)$ since the set $\{\bar{x}, y, z\}$ is a superset of the set $\{\bar{x}, z\}$. Whenever $(\bar{x} \vee z)$ is true, $(\bar{x} \vee y \vee z)$ must also be true.

We have already seen an application of subsumption earlier: During unit propagation, when a unit clause (l) is found, all the clauses containing the literal l are removed from the formula. A corner case is the empty clause, which subsumes all other clauses in a formula.

9.2.4.1. Forward Subsumption

Forward subsumption starts with a given clause C and a formula F , and checks if C is subsumed by F . A simple technique to check if a clause is forward subsumed by a formula F temporarily marks all literals of C as false and then checks whether there is another (usually smaller) clause D in F with only false literals.

In order to find all potential candidates D without iterating over F , at least one literal of every clause in F needs to be “watched”. Two-watched literal schemes [ZS00, MMZ⁺01], as commonly used in CDCL solvers (see Chapter 4), suffice too, while *occurrence lists* (an occurrence list contains *all* the clauses in which a literal occurs) are not necessary. A similar technique has been described in [Bra04] and refined in [Zha05].

9.2.4.2. Backward Subsumption

In contrast to forward subsumption, which checks if a clause C is subsumed by a clause D in F , backward subsumption checks if F contains clauses C that are subsumed by a given clause D , and if so, removes them. Backward subsumption is actually more common in practice; for example, in SAT solvers using clause learning, learned clauses are never forward subsumed, but they might subsume other clauses of the formula, in particular clauses that were learned recently.

As described in Exercise 271 of [Knu15] and its solution, this situation might occur frequently for certain formulas, and it is possible to *eagerly subsume recently learned clauses* efficiently, by reusing the markings of the literals in the learned clause at the end of conflict analysis (see Chapter 4 on CDCL again). Even though this technique is efficient and simple to implement, subsumption algorithms are more useful in combination with variable elimination (see Section 9.3), because variable elimination focuses on irredundant original clauses and is allowed to ignore redundant learned clauses [JHB12].

If an implementation maintains for each literal an occurrence list instead of using a one-watched [Zha05] or two-watched [ZS00, MMZ⁺01] literal scheme, then the search for a candidate clause C , checked for being backward subsumed by the given clause D , can be restricted to iterate over the occurrences of a single literal in the given clause D , chosen as one with the smallest number of occurrences (shortest occurrence list). This observation, originally made in the context of QBF solving [Bie04], is crucial for scaling backward subsumption to large formulas, and was first used for SAT preprocessing in SATELITE [EB05].

Another technique to improve the performance of backward subsumption in practice is based on so-called *signatures* [Bie04], which are an instance of Bloom filters [Blo70]. The basic idea is to avoid costly traversals of occurrence lists in the (frequent) case where no subsumed clauses can be found. This is achieved by checking a signature condition that must hold if there is a clause that can be subsumed by the given clause. If the condition does not hold, the traversal can be avoided. For details, we refer to the original paper.

A related backward-subsumption algorithm is due to Brafman [Bra04]. Given a clause D and a formula F , the algorithm iteratively computes the set

$$S = \bigcap_{l \in D} \text{clauses}_F(l),$$

where $\text{clauses}_F(l)$ denotes the set of all clauses of F in which the literal l occurs. The final set S contains all clauses of F that are subsumed by D . As soon as the algorithm detects that S is empty, it can conclude that D does not subsume any clauses. The original algorithm does not need to save signatures for clauses and literals, but it needs to save the occurrence lists for all the literals in F . In principle, though, it could also make use of signatures.

Backward subsumption can easily be turned into an algorithm that removes all subsumed clauses from a formula: start with the empty formula and one-by-one add original clauses with largest clauses first. Then check for each newly added clause whether it subsumes any of the previous clauses. This observation is particularly useful for applying subsumption during preprocessing. In practice,

it is still advisable to limit the effort spent on subsumption checks. This can, for instance, be done by limiting the maximum length of traversed occurrence lists.

9.2.4.3. Self-Subsuming Resolution

As observed in [EB05], subsumption is closely related to *strengthening* clauses, more precisely to *self-subsuming resolution* as implemented in SATELITE [EB05]. Consider two clauses $C \vee l$ and $D \vee \bar{l}$, which can be resolved upon l . If D subsumes C , then the resulting resolvent is simply C , which in turn subsumes the first antecedent $C \vee l$. Thus, instead of adding the resolvent C , we can replace $C \vee l$ by C , in effect strengthening the clause $C \vee l$ by removing the literal l . This technique is called self-subsuming resolution.

Subsumption algorithms can easily be extended to perform self-subsuming resolution. The only necessary change is to modify the subsumption check between two clauses: when checking if D subsumes C , we simply allow that at most one literal \bar{l} of D occurs negated in C . If this is the case, and all other literals of D occur in C , we can apply self-subsuming resolution to C and D by removing l from C . Checking that the literals of D occur in C (with one possible exception) can be achieved by marking literals (possibly with time stamps) or by a merge-sort-style check, assuming the literals in clauses are sorted [BP11].

Self-subsuming resolution was independently discovered as *global subsumption* for instantiation-based first-order theorem proving [Kor08] and actually was introduced in the SAT context via the notion of a *subsuming resolvent* in [OGMS02] (though without detailed experiments). In [EB05], it was partially motivated by the fact that clause learning (Chapter 4), and particularly clause minimization [SB09], implicitly rely on it. *On-the-fly subsumption* [HJS10, HS09] is another related technique that uses self-subsuming resolution to strengthen antecedent clauses used for deriving learned clauses in CDCL.

9.2.4.4. Implementations

Finding subsumed clauses in CNF formulas is closely related to finding *extremal sets* in data mining. In this context, Bayardo and Panda [BP11] showed that backward-subsumption algorithms developed in the SAT community [EB05] were competitive for this problem too, but their paper also presents a substantially faster algorithm, which in our setting translates to forward-subsumption checking and yields a substantial performance improvement also for SAT.

The basic idea is to use a one-watched-literal scheme and only watch literals with the fewest occurrences. Clauses are added again one-by-one, but now starting with the smallest clauses first. All one-watched-literal lists of the literals in a processed clause are traversed to find clauses which subsume the processed clause. If this is not the case (i.e., the forward-subsumption check fails), then the clause is added, and the literal with the shortest watch list (at this point) is watched. As already discussed above, the actual subsumption check between a previously added clause and the processed clause can be made faster too (for instance, by sorting literals in clauses [BP11]) and extends to self-subsuming resolution.

In the context of QUANTOR [Bie04], SATELITE [EB05], and MINISAT [ES06] backward subsumption was considered superior. However, forward-subsumption



Figure 1. Hypergraph Representation of a CNF Formula.

algorithms inspired by ideas of [BP11], as implemented in SPLATZ [Bie16] and CADICAL [Bie17], become more important, particularly during the process of removing from a given CNF all subsumed clauses and performing self-subsuming resolution until completion. These forward-subsumption algorithms are much more efficient than previously used backward-subsumption algorithms and can even regularly be applied to learned clauses during inprocessing too. Interleaved with variable elimination (see Section 9.3), which needs full occurrence lists in any case, there is still a benefit in using backward subsumption, since it allows to focus on trying to subsume or strengthen other clauses by newly derived clauses to trigger new variable-elimination attempts (as for instance in [BFFH20]).

9.2.5. Connected Components

The solving process for a formula can be significantly simplified by taking its underlying graph structure into account. One way to represent a formula as a graph is to define a hypergraph where variables correspond to vertices and clauses correspond to hyperedges connecting their variables. For example, the formula

$$(x \vee y \vee \bar{z}) \wedge (\bar{y} \vee z) \wedge (u \vee \bar{v}) \wedge (v \vee \bar{w})$$

can be represented by the hypergraph shown in Figure 1. As we can see in this example, the graph contains two different connected components.

It has been observed that formulas in practical SAT solving often contain multiple connected components [BS06]. Each of these components can be seen as an independent SAT problem, with the whole formula being satisfiable only if all of its components are satisfiable. Splitting a formula into its connected components during preprocessing and then solving these components independently has several advantages: the independent components can be solved in parallel, and as soon as one component is identified as unsatisfiable, it can be concluded that the whole formula is unsatisfiable. Even if all components are satisfiable, the parallelism can speed up the solving process. Similar ideas are used in component caching for model counting [BDP03, SBB⁺04], where the formula is preprocessed at every search node and decomposed into disconnected components for which the model count can be computed independently. The problem of model counting is also called #SAT and extensively covered in Chapter 25 and 26.

In the context of CDCL SAT solving (Chapter 4), formulas consisting of independent components can be simpler to solve even without relying on parallelism. Suppose a typical CDCL solver tries to find a satisfying assignment for a formula that consists of several independent components. During search, the solver might find an assignment that already satisfies some of the components, but then realize that the assignment falsifies another component. What the solver usually does in

that situation is to backjump and thereby undo some of its variable assignments. However, this could undo some of the assignments that are independent of the falsified component and thereby unassign components that were already satisfied.

By splitting up a formula, such a situation can be avoided. However, the same effect can be achieved by saving the last assigned truth value (also called *phase*) of a variable and then always assigning that saved value when making decisions. This *phase saving* technique was introduced in the RSAT solver [PD07] and has been standard in SAT solvers since then, making explicit component decomposition obsolete, at least for sequential plain CDCL solving.

Since 2018 several state-of-the-art SAT solvers quite frequently reset saved phases [Bie18, Bie19, BFFH20, ZC20, SCD⁺20, SSK⁺20], which counteracts the effect of phase saving to remember satisfying assignments for components. As an alternative to explicit component decomposition as in [BS06], it was proposed in [BFFH20] to simply remove all clauses satisfied by the largest autarky within saved phases following ideas in [KHB19], an application of autarky reasoning of Chapter 14 within SAT solving.

9.3. Resolution-Based Preprocessing

We continue by discussing preprocessing techniques that rely heavily on resolution, starting with bounded variable elimination, which as high-lighted already in the introduction is still considered the most important preprocessing technique in practical SAT solving.

9.3.1. Bounded Variable Elimination

Bounded variable elimination [Bie03, Bie04, SP04, EB05] is based on the technique of *clause distribution*, which lies at the core of the original Davis-Putnam procedure [DP60]. To perform clause distribution, we choose a variable, add all resolvents upon this variable to the formula, and remove the original clauses containing the variable.

Example 5. Consider the formula

$$F = (x \vee e) \wedge (y \vee e) \wedge (\bar{x} \vee z \vee \bar{e}) \wedge (y \vee \bar{e}) \wedge (y \vee z). \quad (9.1)$$

To perform clause distribution with the variable e , we first add all resolvents upon e . The clauses $(x \vee e)$ and $(y \vee e)$ can both be resolved with $(\bar{x} \vee z \vee \bar{e})$ and $(y \vee \bar{e})$. We thus add the corresponding four resolvents to obtain the formula

$$F \wedge (x \vee \bar{x} \vee z) \wedge (x \vee y) \wedge (y \vee \bar{x} \vee z) \wedge (y).$$

Now we remove all clauses that contain e to obtain

$$(y \vee z) \wedge (x \vee \bar{x} \vee z) \wedge (x \vee y) \wedge (y \vee \bar{x} \vee z) \wedge (y). \quad (9.2)$$

Repeated clause distribution can increase the number of clauses exponentially, rendering its unbounded use as a preprocessing technique unaffordable. Actually, already eliminating a single variable might increase the size quadratically. To

deal with this problem, the original technique of bounded variable elimination applies clause distribution only on variables whose elimination does not increase the number of clauses, which explains the name *bounded* variable elimination.

However, as proposed in the variable-elimination procedure of the QBF solver QUANTOR [Bie04], it has become common in SAT to relax the bound on additional clauses: in *incrementally relaxed bounded variable elimination*, introduced in the SAT solver GLUEMINISAT [NII15], the bound is increased³ incrementally every time a round of variable elimination has completed without increasing the size of the formula too much. This technique was ported to the highly influential SAT solver COMINISATPS [Oh16] that has formed the basis of the MAPLESAT series of SAT solvers [LGPC16, LLX⁺17, NR18], which ranked at the top in the SAT competitions from 2016 to 2018.

Since bounded variable elimination produces many redundant clauses, it is often combined with tautology elimination (as in NIVER [SP04]) as well as with subsumption and strengthening (in SATELITE [EB05]). Subsumption might lead to further variables being eliminated, triggered by removing clauses or literals.

Example 6. Consider again the formula (9.2), which resulted from the formula (9.1) by eliminating the variable e . We can observe that the resolvent $(x \vee \bar{x} \vee z)$ is a tautology and that it can therefore be removed from the formula. Moreover, the clauses $(y \vee z)$, $(x \vee y)$, and $(y \vee \bar{x} \vee z)$ are all subsumed by the clause (y) , and so they can be eliminated as well. Thus, after eliminating only a single variable, we end up with the trivially satisfiable formula (y) .

On the implementation side, clause distribution is best interleaved with subsumption using the generated resolvents immediately to subsume and strengthen existing clauses through backward subsumption. Furthermore, bounded variable elimination can be improved by on-the-fly subsumption during variable elimination [HS09] as follows: Whenever a (non-tautological) resolvent R of two clauses C and D is computed, check if $|R| = |C| - 1$ or $|R| = |D| - 1$ (after removing repeated literals, i.e., interpreting clauses as literal sets). In the former case, C is subsumed by R and so it can be replaced by R ; in the latter case, D is subsumed by R and so it can also be replaced by R . Moreover, the replacement can be performed regardless of whether e is actually eliminated or not (in case elimination of e would produce too many new clauses). Variables of removed clauses or literals have to be (re)scheduled as candidates for further elimination attempts.

Another important implementation detail is how to organize the schedule of elimination candidates. Common practice is to use a priority queue implemented as a binary heap in which the variables are ordered by their number of occurrences within the formula. Variables with the smallest number of occurrences are tried first, and the priority queue is updated dynamically as clauses are strengthened, removed, and added. In order to scale variable elimination to large formulas, variables that—when resolved—produce large resolvents or occur in large or in too many clauses, should not be eliminated. Limits vary by implementation, but typically the clause size or the resolvent size is limited to 20 to 100 literals. Variables that occur more than 100 to 1000 times, either negatively or positively, are typically skipped too.

³For instance, the bounds form a geometric series 0, 8, 16, 32, . . . , 8192.

The effectiveness of bounded variable elimination is highlighted by the observation that it automatically performs the earlier discussed elimination of pure literals, because there are no resolvents upon pure literals. Moreover, if a formula contains a unit clause (e) and either on-the-fly subsumption is performed or the formula contains no clauses that are subsumed by (e), then the result of eliminating the variable e is the same as applying the unit-clause rule with (e).

As observed in [EB05], the elimination of variables for which the CNF contains a functional definition produces redundant clauses; these clauses are actually resolvents of clauses that are not part of the functional definition. Thus, searching for such definitions (see Section 9.6.2) during elimination attempts reduces the number of required resolvents and often allows more variables to be eliminated.

Alternatively, as implemented in LINGELING [Bie10], it is possible to simplify the CNF of resolvents on the fly, for instance by applying “semantically” the dual of Minato’s algorithm for producing an irredundant sum-of-products [Min92] to CNF [EMS07]. The idea is to compile the CNF of the clauses containing the variable to be eliminated into a “semantic” representation, such as a binary decision diagram (BDD) [Bry86], eliminate the variable from the BDD by existential quantification, and then as in [EMS07] encode the resulting BDD back into an irredundant CNF (using Minato’s algorithm). Instead of using BDDs *function tables* represented as bit-maps as in LINGELING will also do. In any case, this approach avoids searching for functional definitions explicitly and captures all types of functional dependencies. Otherwise it reduces the number of produced clauses at least as much as the explicit method [EB05] discussed above.

For the sake of completeness, it should also be mentioned that there exist symbolic variants of bounded variable elimination that use decision diagrams [CS00, vDEB18], more precisely *zero-suppressed decision diagrams* (ZDD) [Min93]; but in practice they currently work only on very restricted sets of formulas (e.g., pigeon-hole formulas).

Finally, bounded variable elimination can also be applied “in reverse”, resulting in what is called *bounded variable addition* (BVA), as proposed in [MHB12]. Since the elimination of variables can increase the number of clauses, performing variable elimination in reverse (adding variables instead of eliminating them) can potentially shrink the size of a formula. Requiring to distinguish between two types of variables (i.e., original and added) makes the implementation of bounded variable addition much more problematic than implementing bounded variable elimination, particularly in the context of incremental SAT solving, where also the need of “recycling” variables might become an issue. In contrast to bounded variable elimination, bounded variable addition has therefore not yet established itself as a central preprocessing technique.

9.3.2. Techniques Based on Implication Graphs

Analyzing the structure of binary clauses (i.e., clauses that contain exactly two literals) in a CNF formula gives rise to preprocessing techniques through the notion of a (*binary*) *implication graph* [APT79] (BIG). Given a CNF formula, its binary implication graph is a directed graph that is obtained as follows: Introduce a vertex for every literal, and add the edges (\bar{l}, k) and (\bar{k}, l) for every binary clause $(l \vee k)$, where l and k are (possibly negative) literals.

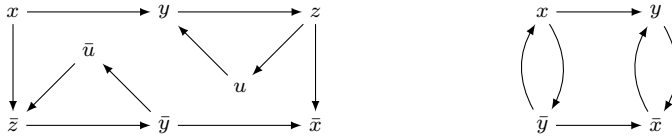


Figure 2. The original (binary) implication graph (BIG) for the formula of Example 7 is shown on the left and on the right the result after one round of equivalent-literal substitution.

Example 7. The implication graph of the formula

$$(\bar{x} \vee y) \wedge (\bar{y} \vee z) \wedge (\bar{z} \vee u) \wedge (\bar{u} \vee y) \wedge (\bar{x} \vee \bar{z}) \wedge (x \vee z \vee u)$$

is shown on the left in Figure 2. Notice that the ternary clause $(x \vee z \vee u)$ does not affect the implication graph.

An edge in the implication graph directly represents an implication that is equivalent to a binary clause in the formula. In the example, x implies y , y implies z , z implies u , and so on. Notice also that for every edge of the form (e_1, e_2) , the implication graph contains a corresponding edge (\bar{e}_2, \bar{e}_1) , representing the contraposition of the implication.

By construction, every *strongly connected component*⁴ of a binary implication graph represents a set of *equivalent literals* [VGT93, Li00], i.e., literals who must have the same truth value in every satisfying assignment of the formula. The set of literals prescribed by a strongly connected component can therefore be replaced by a single representative literal in the underlying CNF formula; this replacement is known as *equivalence reduction* and *equivalent-literal substitution* (ELS).

Example 8. In the implication graph on the left in Figure 2, the literals y , z , and u form a strongly connected component. Their negations \bar{y} , \bar{z} , and \bar{u} also form a strongly connected component. Since a clause $(x \vee y)$ is represented by the two edges (\bar{y}, x) and (\bar{x}, y) in the binary implication graph, every strongly connected component has a corresponding strongly connected component that consists of its complementary literals. In the example, we can replace the strongly connected component $\{y, z, u\}$ by the vertex y , and the strongly connected component $\{\bar{y}, \bar{z}, \bar{u}\}$ by the vertex \bar{y} , to obtain the simplified formula

$$(\bar{x} \vee y) \wedge (\bar{x} \vee \bar{y}) \wedge (x \vee y).$$

Note that the clause $(x \vee y)$ stems from the original ternary clause $(x \vee z \vee u)$. The simplified formula corresponds to the implication graph depicted on the right of Figure 2 (with two new strongly connected components).

There are several different implementations for finding and replacing strongly connected components [APT79, dV01, BW03, VG05, GS05] in the context of SAT. For details, we refer to the respective literature, but usually the implementations are based on a depth-first search of the implication graph.

⁴A strongly connected component is a maximal set of vertices such that every vertex in the set is reachable from all other vertices in the set.

In certain cases, strongly connected components can even be used for detecting unsatisfiability: if both a literal l and its complement \bar{l} are contained in the same strongly connected component, then the formula is unsatisfiable. Additionally, the implication graph can be used for the detection of failed literals. In particular, if there is a path from a literal l to its complement \bar{l} , then l is a failed literal [VG05].

Example 9. In the binary implication graph shown in Figure 2, x is a failed literal because there is a path from x to \bar{x} . Note, however, that the implication graph does not always contain a path from a failed literal l to its complement \bar{l} . To see this, consider the formula $F = (\bar{x} \vee y \vee z) \wedge (\bar{x} \vee \bar{y}) \wedge (\bar{x} \vee \bar{z})$ and the literal x . Although it can be checked that x is a failed literal with respect to F , there is no path from x to \bar{x} in the implication graph, because the ternary clause $(\bar{x} \vee y \vee z)$, without which x would not be a failed literal, is not considered when constructing the implication graph.

9.3.3. Hyper Binary Resolution

We have already seen in the context of unit propagation how a formula can be simplified by continuously performing inference steps (namely, the *unit-clause rule*) until either a conflict is derived or no more inference steps are applicable. Hyper binary resolution is based on the same idea, but uses stronger inferences.

Towards hyper binary resolution, consider first *binary resolution*. The binary-resolution rule is obtained from the ordinary resolution rule by restricting it to binary clauses. A binary-resolution step can either produce another binary clause—for instance, if we resolve $(x \vee y)$ with $(\bar{x} \vee z)$ to obtain $(y \vee z)$ —or it can yield a unit clause—for instance, if we resolve $(x \vee y)$ with $(\bar{x} \vee y)$ to obtain (y) . Applying the binary-resolution rule and the unit-clause rule until none of them is applicable anymore can simplify a formula more aggressively than performing only unit propagation.

Example 10. Let $F = (x \vee y) \wedge (\bar{x} \vee y) \wedge (x \vee \bar{y}) \wedge (\bar{x} \vee \bar{y})$. As there are no unit clauses in F , unit propagation has no effect on the formula. However, by performing binary resolution of $(x \vee y)$ with $(\bar{x} \vee y)$, we derive (y) . After this, unit propagation can derive a conflict with the clauses (y) , $(x \vee \bar{y})$, and $(\bar{x} \vee \bar{y})$.

As observed by Bacchus [Bac02], every unit clause that can be derived by performing binary resolution in combination with unit propagation (as described above) can also be derived by failed-literal detection, but not vice versa.

Example 11. With respect to the formula $F = (\bar{x} \vee y \vee z) \wedge (\bar{x} \vee \bar{y}) \wedge (\bar{x} \vee \bar{z})$ from Example 9 above, the literal x is a failed literal since unit propagation derives a conflict on $F \wedge (x)$. Therefore (\bar{x}) can be derived by failed-literal detection. However, both unit propagation and binary resolution do not affect F . The reason for this is that they cannot use the first clause, which contains three literals.

Simply performing all resolutions, also with longer clauses, would produce too many clauses. This leads to the definition of hyper binary resolution [Bac02], an instance of *hyper resolution* [Rob74], which contracts several resolution steps into a single inference step.

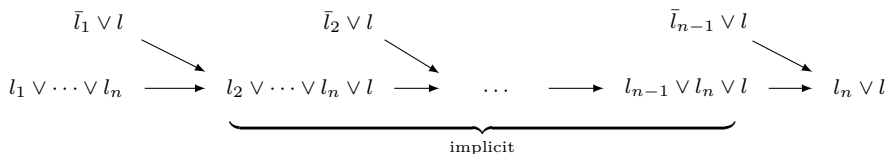


Figure 3. Hyper binary resolution. Only the last resolvent ($l_n \vee l$) is actually added after a hyper binary resolution step. The intermediate resolvents are implicit.

Definition 12. Given an n -ary clause $(l_1 \vee \dots \vee l_n)$ and $n - 1$ binary clauses $(\bar{l}_1 \vee l), \dots, (\bar{l}_{n-1} \vee l)$, *hyper binary resolution* allows the derivation of the clause $(l_n \vee l)$, which is called a *hyper binary resolvent*.

Observe that the hyper binary resolvent ($l_n \vee l$) can be derived from the previous clauses by applying a number of resolution steps. We start with $(l_1 \vee \dots \vee l_n)$ and first resolve it with $(\bar{l}_1 \vee l)$ to obtain $(l_2 \vee \dots \vee l_n \vee l)$, which we then resolve with $(\bar{l}_2 \vee l)$ to obtain $(l_3 \vee \dots \vee l_n \vee l)$ and so on until we finally derive $(l_n \vee l)$. The intermediate resolvents, however, are not derived explicitly. Figure 3 illustrates the (implicit) resolutions involved in a single hyper binary resolution step.

By repeatedly performing hyper binary resolution in combination with unit propagation until no more inference steps are possible (or a conflict is derived), a formula can be simplified significantly. In fact, this combination derives the same literals that would be derived by performing failed-literal detection on all literals in combination with unit propagation [Bac02]. It can, for instance, derive the literal (\bar{x}) from the formula F in Example 11 by a single application of the hyper binary resolution rule.

Hyper binary resolution and unit propagation have been used in combination with the detection of binary equivalences (replacing them by single literals, similar to the technique based on strongly connected components discussed in Section 9.3.2). The resulting preprocessing technique can be implemented based on probing failed literals in a depth-first search of the binary implication graph [BW03]. As soon as a large (non-binary) clause is used to derive a unit clause during probing, a hyper binary resolvent is learned. A less general but faster version of this technique, which only probes roots of the binary implication graph, has been implemented by Gershman and Strichman [GS05].

More sophisticated algorithms for hyper binary resolution [HJB13] reuse decisions and propagation effort by scheduling decisions along the structure of the binary implication graph, but using multiple decisions at the same time, similar in spirit to efficient “distillation” on tries [HS07], with more details provided at the end of the next section.

These approaches avoid “transitive” hyper binary resolvents by prioritizing unit propagation over binary clauses. For instance, the set $\{(\bar{a}_i \vee a_{i+1}) \mid 1 \leq i < n\}$ of binary clauses has quadratically many hyper binary resolvents $(\bar{a}_i \vee a_j)$, with $1 \leq i < j \leq n$. Note that these additional transitive clauses are redundant with respect to detecting strongly connected components of equivalent literals as well as unit propagation (see also the discussion of “empowering” in [PD11]) and thus should not be derived (and stored). It is further useful to apply transitive reduc-

tion of the binary implication graph pro-actively [HJB10a]. As first implemented in PRECOSAT [Bie09] and further refined in [HJS11], there is also an on-the-fly variant of hyper binary resolution that learns binary clauses during propagation using dominator analysis on the binary implication graph (see also [HJB13]).

However, removing transitive resolvents does not prevent the worst case of quadratically many non-transitive hyper binary resolvents [HJB13], and running hyper binary resolution until completion is only feasible for certain formulas (mostly small and hard combinatorial problems). For large application formulas, hyper binary resolution has to be preempted, and it is advisable to “forget” hyper binary resolved clauses aggressively [BFFH20] during the “reduce” phase of the SAT solver which deletes useless learned clauses.

9.3.4. Advanced Probing Techniques

We start with asymmetric⁵ tautologies, which are a generalization of tautologies based on the concept of an *asymmetric literal*:

Definition 13. A literal l is an *asymmetric literal* in a clause $C \vee l$ with respect to a formula F if unit propagation deduces the unit clause \bar{l} from $F \wedge \bar{C}$, where \bar{C} is the conjunction of the negations of the literals in C .

Accordingly, if l is an asymmetric literal in $C \vee l$ w.r.t. F , then all models that falsify C but satisfy F , falsify l . In this case, l is redundant in $C \vee l$ and $F \wedge (C \vee l)$ can be simplified to $F \wedge C$ by “strengthening” $C \vee l$ and removing l . This simplification is also called *asymmetric literal elimination* (short ALE).

Example 14. Consider the formula $F = (a \vee \bar{x}) \wedge (x \vee \bar{l})$ and the clause $C = (a \vee b)$. Unit propagation on $F \wedge \bar{C} = F \wedge (\bar{a}) \wedge (\bar{b})$ produces the unit clause (\bar{l}) . Therefore, l is an asymmetric literal in $C \vee l$ with respect to F , and $(a \vee \bar{x}) \wedge (x \vee \bar{l}) \wedge (a \vee b \vee l) = F \wedge (C \vee l)$ can be simplified to $(a \vee \bar{x}) \wedge (x \vee \bar{l}) \wedge (a \vee b) = F \wedge C$ through asymmetric literal elimination.

In order to deduce that clauses are “redundant” based on this notion, we add literals instead of removing them, and during this process obtain a tautological clause [HJB10a].

Definition 15. A clause C is an *asymmetric tautology* (AT) with respect to F if there is a sequence l_1, \dots, l_n of literals such that $C \vee l_1 \vee \dots \vee l_n$ is a tautology and each l_i is an asymmetric literal in $C \vee l_1 \vee \dots \vee l_i$ with respect to F .

Example 16. Consider the formula $F = (\bar{x} \vee z) \wedge (\bar{y} \vee \bar{z})$ and the clause $C = (\bar{x} \vee \bar{y})$. The literal \bar{z} is an asymmetric literal in $(\bar{x} \vee \bar{y} \vee \bar{z})$ w.r.t. F since unit propagation on $F \wedge (x) \wedge (y)$ deduces the unit clause (z) (propagation of $F \wedge (x)$ is enough). Then z is an asymmetric literal in $(\bar{x} \vee \bar{y} \vee \bar{z} \vee z)$ w.r.t. F as unit propagation deduces also \bar{z} from $F \wedge (x) \wedge (y) \wedge (z)$ (again already unit propagation of $F \wedge (y)$ suffices). At this point we can stop, since we have already shown that C in combination with F is equivalent to a tautology, thus an asymmetric tautology.

⁵The term “asymmetric” has its origin in the notion of “asymmetric (tableau) rules” [DM94], which apparently inspired the MINISAT authors [ES03] to name a command line option “-**asymm**” used to enable a “strengthening” procedure now called “asymmetric literal elimination”.

Note that in this example we used the fact that unit propagation is confluent in the sense that the order in which the unit-clause rule is applied is irrelevant to the outcome; it is also monotonic in the sense that every unit clause that can be derived from a formula F can also be derived from every larger formula that contains all the clauses of F .

It can further be shown that if a clause C is an asymmetric tautology w.r.t. a formula F , then F implies C . This in turn means that C can be safely eliminated from the formula $F \wedge C$. Moreover, if only asymmetric literals from binary clauses need to be added to turn a clause into a tautology, then that clause is also called a *hidden tautology*, and the corresponding clause-elimination technique is called *hidden tautology elimination* (HTE) [HJB10a].

The related technique of *unhiding* [HJB11] avoids quadratic computation during repeated unit propagation. It is based on randomized depth-first traversal of the binary implication graph and relies on the *parenthesis theorem* to detect in almost constant time if a clause is a hidden tautology or if clauses can be strengthened by removing hidden literals. This form of literal removal is also known as *hidden literal elimination* (HLE), which can be seen a restricted variant of asymmetric literal elimination which only propagates over binary clauses.

It is well-known that a clause C is an asymmetric tautology with respect to a formula F if and only if it is a *reverse unit propagation* (RUP) clause [VG12].

Definition 17. A clause C is a RUP clause with respect to a formula F if unit propagation derives a conflict on $F \wedge \bar{C}$.

To test if a clause is an asymmetric tautology with respect to a formula F , it thus suffices to check if propagating its negation leads to a conflict. Note that RUP clauses are a generalization of the earlier-mentioned failed literals: a unit clause (l) is a RUP clause with respect to a formula F if and only if \bar{l} is a failed literal with respect to F . From the RUP definition it also becomes clear why asymmetric tautologies are implied, since unit propagation can only derive a conflict if the conjunction of the formula with the negated clause is unsatisfiable, which is the case if and only if the clause is implied by the formula.

Failed-literal probing alone is often already quite expensive. Thus checking all clauses for being asymmetric tautologies is even more costly. Still, removing them can benefit other preprocessing algorithms such as variable elimination.

The basic approach of asymmetric tautology checking takes a clause C , then assigns all its literals to false, followed by unit propagation. Advanced probing techniques like *distillation* [HS07] and *vivification* [PHS08] interleave assignments and propagations instead. This works as follows.

The literals of C are still assigned to be false in an arbitrary fixed order, but a complete round of propagation is started immediately after every single assumed assignment. If during such a propagation round a conflict is found or another not yet assigned literal of C is forced to true, then the clause is an asymmetric tautology and can in principle be removed. Alternatively, particularly if applied to learned clauses in CDCL [LLX⁺17, LXL⁺20], such asymmetric tautologies can be replaced by clauses learned through the standard CDCL conflict analysis, if, for instance, the learned clause turns out to be shorter. If, however, during propagation another not yet assigned literal of C is forced to false, then this

literal is actually an asymmetric literal and the clause C can be strengthened, by removing the literal from C . This presents the real benefit of interleaving assignments and propagation in vivification and distillation.

A disadvantage of clause-based probing, in contrast to literal based probing discussed before, is that potentially many redundant propagations are performed, for instance when many clauses share the same literals. These literals are then repeatedly assigned to false and propagated. An attempt to reduce this redundant propagation effort was made in distillation by reorganizing the clauses in a trie (i.e., as a shared prefix tree) and reusing propagations along the same prefix [HS07]. The implementation of vivification in CADICAL [Bie17] and KISSAT [BFFH20] achieves the same effect by sorting clauses and literals, thus simulating a trie structure on a plain CNF formula.

Another important question is which clauses should be checked, particularly if these techniques are applied to learned clauses during inprocessing [JHB12]. One proposal was to check learned clauses in parallel [WH13] in a separate thread. Another, which was actually the basis for the revival of these techniques [LLX⁺17], was to focus on learned clauses, check all clauses at most once, and check only those which have a high chance of being used (e.g., clauses of small LBD [AS09]). For more details see [LXL⁺20].

9.4. CNF Preprocessing Beyond Resolution

So far, we have only considered the elimination of clauses that are implied. However, as we discussed in the section on subsumption, in SAT solving often a more general notion of redundancy is used, requiring only that the elimination of a clause has no effect on the satisfiability status of a formula (see [JHB12]).

One of the most important types of clauses that are redundant but not necessarily implied, are *blocked clauses*. Intuitively, a clause is blocked if it contains a literal such that all resolvents upon this literal are tautologies [Kul99].

Definition 18. Given a formula F and a literal l , we denote by F_l the set of clauses of F that contain l . Then a clause C is a *blocked clause* in a formula F if it contains a literal l such that for every clause $D \in F_{\bar{l}}$, the resolvent of C and D upon l is a tautology. We say that l blocks C in F .

Example 19. Consider the formula $F = (\bar{x} \vee z) \wedge (\bar{y} \vee \bar{x})$ and the clause $(x \vee y)$. The literal x does not block $(x \vee y)$ in F since the resolvent $(y \vee z)$ of $(x \vee y)$ and $(\bar{x} \vee z)$ is not a tautology. However, the literal y blocks $(x \vee y)$ in F since the only clause in $F_{\bar{y}}$ is the clause $(\bar{y} \vee \bar{x})$, and the resolvent $(x \vee \bar{x})$ of $(x \vee y)$ and $(\bar{y} \vee \bar{x})$ is a tautology. Therefore, $(x \vee y)$ is blocked in F .

Blocked clauses were initially introduced by Kullmann [Kul99] as a generalization of the definition clauses in extended resolution [Tse68], and were later proposed as a basis for the preprocessing technique of blocked clause elimination [JBH10] (BCE). As further shown in [JBH12], blocked clause elimination can simulate several structural simplification techniques (see also Section 9.6) on the CNF-level, in particular *non-shared-input elimination*, *monotone-input reduction*, and *cone-of-influence reduction*. Moreover, the effectiveness of bounded variable elimination can be increased by interleaving it with blocked clause elimination.

Blocked clauses have been generalized in several ways. One well-known generalization are *resolution asymmetric tautologies*, better known as RATs. The RAT definition is obtained from the blocked-clause definition by replacing *tautologies* with *asymmetric tautologies* [JHB12].

Definition 20. A clause C is a *resolution asymmetric tautology* (RAT) in a formula F if it contains a literal l such that for every clause $D \in F_l$, the resolvent of C and D upon l is an asymmetric tautology with respect to F .

From this definition it immediately follows that every blocked clause in formula F is also a RAT with respect to F . Although the explicit elimination of RATs has not yet been shown to boost solver performance in SAT, generalizations of RAT elimination for quantified Boolean formulas (QBFs) have led to significant performance improvements in QBF solving [LE18].

Moreover, RAT lies at the core of the well-known DRAT proof system, which is the de-facto standard in modern SAT solving. Other generalizations of blocked clauses and RATs (e.g., *set-blocked clauses* [KSTB16, KSTB18] and *propagation-redundant clauses* (PR) [HKB17, HKB20]) have also not yet been used for clause elimination but instead for clause addition during solving and for proof generation, which we are not going to discuss here.

One type of clauses that generalizes blocked clauses, and whose elimination has shown some performance improvements in the SAT solver LINGELING [Bie17], are *covered clauses*. Intuitively, a clause is a covered if it can be turned into a blocked clause by adding so-called *covered literals* [HJB10b].

Definition 21. A literal k is a *covered literal* in a clause C with respect to a formula F if C contains a literal l such that all non-tautological resolvents of C upon l with clauses D in F_l contain k .

The addition of covered literals preserves satisfiability in the sense that $F \wedge C$ and $F \wedge (C \vee k)$ are equisatisfiable if k is covered by C in F ; before an example, we give the definition of covered clauses [HJB10b] based on covered literals.

Definition 22. A clause C is a *covered clause* with respect to a formula F if there exists a sequence k_1, \dots, k_n of literals such that each k_i is covered by $C \vee k_1 \vee \dots \vee k_{i-1}$ with respect to F and $C \vee k_1 \vee \dots \vee k_n$ is blocked in F .

Example 23. Consider the formula $F = (\bar{x} \vee y) \wedge (\bar{y} \vee z)$ and the clause $C = (x \vee \bar{z})$. Although C is not blocked in F , we can add the literal y since it is contained in the only resolvent upon x , namely $(y \vee \bar{z})$, obtained by resolving with $(\bar{x} \vee y)$. The resulting clause $(x \vee \bar{z} \vee y)$ is then blocked as there is only the tautological resolvent $(x \vee \bar{z} \vee z)$ upon y , obtained by resolving with $(\bar{y} \vee z)$. Thus, C is a covered clause with respect to F .

There exist even more general types of redundant clauses, such as *resolution-subsumed clauses* [JHB12] (“the clause contains a literal such that all resolvents upon the literal are tautological or subsumed”), and asymmetric variants of blocked as well as covered clauses (“the addition of asymmetric or covered literals turns the clause into a blocked clause”) [HJL⁺15, JHB12], thus extending *blocked clause elimination* (BCE) and *covered clause elimination* (CCE) to the more general concept of *asymmetric covered clause elimination* (ACCE).

It is, however, not clear if the elimination of such clauses can lead to noticeable performance improvements in practical SAT solving. The same applies to the elimination of *globally blocked clauses* [KHB19] motivated by circuit preprocessing techniques. There is also some renewed interest in covered clauses, which were recently shown to have orthogonal strength to propagation-redundant clauses [BCB20].

9.5. Solution Reconstruction

For many practical applications it is not sufficient to regard SAT as a mere decision problem. Consider, for instance, bounded model checking (Chapter 18): If a SAT solver determines that a propositional encoding of a safety property is satisfiable, we know that a bad state is reachable and thus that the safety property is violated. Without a concrete satisfying assignment, however, we cannot obtain an execution trace to analyze and diagnose the reason for the property violation. A similar situation occurs in SAT-based planning (Chapter 19), where a simple yes/no answer without the ability to generate actual plans is (mostly) useless.

Other applications require proofs after the SAT solver determined a formula to be unsatisfiable (e.g., to extract *clausal cores* or to generate *interpolants*). Preprocessors thus also need to make sure that valid proofs of a simplified formula can be turned into valid proofs of the original formula. For clausal proofs, which have been the dominant proof variant in the SAT competition since 2013, this is often easy to achieve [JHB12], and only requires that added clauses can be derived via a sequence of simple proof steps (for many techniques, a simple proof step via a RAT addition often suffices). Producing deletion information, used to speed up proof checking for DRAT proofs [HHJW13, WHHJ14], is slightly more involved, as additional care needs to be taken to avoid that clauses are deleted too early. See Chapter 15 on proofs of unsatisfiability for more details.

In the rest of this section, we focus on satisfiable formulas, and, following the literature [JB10], we use the terms *satisfying assignment*, *model*, and *solution* interchangeably. Moreover, we interpret *solution reconstruction* as the process of turning a satisfying assignment of the simplified formula (i.e., the formula produced by preprocessing) into a satisfying assignment of the original formula.

In order to obtain solution reconstruction for unit propagation (Section 9.2.1), it is common practice to simply keep unit literals permanently assigned. That is also the reason why they are sometimes called *root-level assigned*, *root-level forced* or simply *fixed* literals, where the *root level* is the top-most decision level (where no decisions have been made) in a CDCL solver (Chapter 4). This covers also the effect of failed-literal probing. Solvers might actually contain a global Boolean flag to record that the formula is *root-level inconsistent*, meaning that either the input CNF contains the empty clause or the empty clause has been derived (through preprocessing or in conflict analysis during CDCL search).

In industrial applications, it is not uncommon that preprocessing removes around 80% of the variables [EB05]. Focusing on the remaining “working set” of *active variables* can therefore improve cache efficiency and reduce memory consumption. Variables become inactive during search or preprocessing if they are for instance fixed (or eliminated). Accordingly, several SAT solvers distinguish

internal variables and *external variables* in order to keep the working set compact. External variables are those seen by the user of the (incremental) SAT solver and are mapped (and occasionally remapped during inprocessing) to the remaining internal variables on which the SAT solver works internally. Fixed external variables can either be recorded to be fixed (to a certain Boolean value) in this mapping directly, or can be mapped to a unique internal fixed variable (positively or negated), which represents all the fixed (external) variables.

All forms of strengthening and subsumption considered in this chapter, including vivification and distillation, preserve logical equivalence and thus can be ignored from the perspective of solution reconstruction (for satisfiable formulas). For pure literals solution reconstruction can be achieved by simply adding the corresponding unit clauses (which are RAT). However, as discussed in Section 9.2.3, this method does not preserve logical equivalence, since the formula might also allow models where pure literals are false. This is considered problematic for incremental SAT, since the user could be interested in such models in consecutive SAT queries. An alternative is to use a reconstruction stack as discussed further down, in essence treating pure literals in the same way as eliminated variables.

Equivalent-literal substitution (ELS) (see Section 9.3.2) also only preserves satisfiability, since substituted literals do not occur in the simplified formula anymore and can assume any value. One way of dealing with this situation, as implemented in LINGELING [Bie11], is to maintain a global union-find data structure [Tar79] of substituted literals which maps literals to the representative literal of their equivalent-literal class. The user can then obtain the solution value of a literal from its representative literal. This has the additional benefit that users can rather cheaply query the SAT solver, whether two literals have already been determined to be equivalent. This union-find data structure can also be shared among several solver threads efficiently as in PLINGELING [Bie11]. Fixed literals (units) can be handled within the same scheme by adding a pseudo-representative constant literal (for, say, the Boolean constant *false*).

9.5.1. Reconstruction Stack

Bounded variable elimination, is on the one hand the most important preprocessing technique, but on the other hand does not preserve logical equivalence, by the same argument as for equivalent-literal substitution: after elimination, the eliminated variable is not restricted anymore and can assume any value. Therefore, we really need a method for solution reconstruction.

One way to perform solution reconstruction for variable elimination works as follows. Take the given solution of the simplified formula (restricted to the remaining variables) and add satisfied literals as unit clauses to the original formula. Alternatively, add back the eliminated clauses to the simplified formula plus the satisfied literals of the solution as units to obtain the augmented formula. Then call a second time a (plain) SAT solver on this augmented formula, which is guaranteed to be satisfiable, and that way compute a satisfying assignment of the original formula. The hope is that the additional unit clauses make it easy to solve the augmented formula, which is not guaranteed though. A similar scheme was employed in the SAT preprocessor SATELITE [EB05], using temporary (bi-

nary) files for communicating the simplified CNF and the eliminated clauses to the second SAT solver.

A more sophisticated approach, guaranteed to have even linear complexity in the number of eliminated clauses, which was first described in the literature in [JB10], goes back to Niklas Sörensson. He observed that the given solution of the simplified formula can be “extended” to a solution of the original formula by propagating along the eliminated clauses in a specific way. The corresponding reconstruction algorithm can be described as follows.

Assume that whenever a variable is eliminated, the clauses in which the variable occurs are pushed on a stack, called the *reconstruction stack*. Further, make sure that the first literal in each pushed clause is either the eliminated variable or its negation, in order to map eliminated clauses to eliminated variables.

After obtaining a solution of the simplified formula, it is first extended by assigning an arbitrary value (e.g., false) to all eliminated variables. Then, solution reconstruction goes over the clauses on the reconstruction stack in reverse order, starting from the last pushed clause. Each clause is checked to be satisfied, and if not (i.e., all its literals are assigned to false), then the value of its first literal, which corresponds to the eliminated variable, is flipped (assigned opposite value).

Example 24. Consider the formula $(\bar{a} \vee b) \wedge (a \vee \bar{b}) \wedge (b \vee c) \wedge (\bar{b} \vee \bar{c})$ which is satisfiable and encodes that a and b need to have the same truth value but b and c need to have opposite values. Now assume that variable elimination first eliminates a , then b , and during this process pushes these four clauses in the given order onto the reconstruction stack. The simplified formula consists of the empty CNF and has as extended solution the assignment where all variables are set to false. Now going backward over the reconstruction stack the algorithm first checks whether the last clause $(\bar{b} \vee \bar{c})$ is satisfied, which is the case. The next clause in reverse order $(b \vee c)$ is falsified however. Thus we flip the value of its first literal b and set b to true. The next clause to be checked $(a \vee \bar{b})$ is also falsified (since a is still false and b just became true). Accordingly, we flip the value of its first literal a and set a to true. The last clause to be checked, $(\bar{a} \vee b)$, is satisfied. Thus the reconstructed solution has both a and b set to true (assigned to the same value), while c remains false (assigned differently).

In fact, this simple algorithm also works for more advanced clause elimination procedures: Blocked clauses are just pushed on the reconstruction stack with the blocking literal as first *witness literal*. The same principle extends without modification to more powerful clause redundancy properties [HJB10a, HJB10b, HJL⁺15], including RAT, CCE and ACCE [JHB12], the latter two with the caveat of requiring to potentially push multiple clauses on the reconstruction stack when eliminating a clause [BCB20]. This scheme also allows to handle autarky elimination [BFFH20] as well as equivalent-literal substitution elegantly [FBS19], instead of using a union-find data structure. Finally, solution reconstruction for propagation-redundant clauses [HKB17, HKB20] can be achieved in a similar way, using a reconstruction stack, even for incremental SAT solving [FBS19]. The only change needed, as implemented in CADICAL to support globally blocked clauses [KHB19], is to replace the single witness literal by a set of witness literals, which are all set to true if the corresponding clause is falsified.

9.6. Structure-Based Preprocessing

So far we have focused on preprocessing techniques developed specifically for formulas in conjunctive normal form. We now extend the discussion to preprocessing techniques that were proposed for more expressive constraints and structural representations. In particular, we discuss parity-reasoning techniques and preprocessing techniques applicable on Boolean circuits, which offer a compact representation of arbitrary propositional formulas.

9.6.1. Parity Reasoning

Although the CNF representation of propositional formulas has many advantages, it can be suboptimal in some cases. One such case, which occurs frequently, is when formulas encode so-called *XOR constraints* or *equivalence clauses*. An XOR constraint is of the form

$$l_1 \oplus l_2 \oplus \dots \oplus l_k = b,$$

where l_1, \dots, l_k are propositional literals and b is either 0 or 1. Following the standard semantics of the \oplus operator ($x \oplus y$ is true if and only if x and y are assigned different truth values), an assignment satisfies an XOR constraint with $b = 1$ if and only if it satisfies an odd number of the literals l_1, \dots, l_k . Respectively, it satisfies an XOR constraint with $b = 0$ if it satisfies an even number of the literals. Equivalence clauses are closely related to XOR constraints. An equivalence clause is of the form

$$l_1 \leftrightarrow l_2 \leftrightarrow \dots \leftrightarrow l_k.$$

An assignment satisfies an equivalence clause if and only if it falsifies an even number of the literals l_1, \dots, l_k . Observe that an equivalence clause of the form $l_1 \leftrightarrow \dots \leftrightarrow l_{2i+1}$, having an odd number of literals, is equivalent to the XOR constraint $l_1 \oplus \dots \oplus l_{2i+1} = 1$. Likewise, an equivalence clause $l_1 \leftrightarrow \dots \leftrightarrow l_{2i}$ with an even number of literals is equivalent to the XOR constraint $l_1 \oplus \dots \oplus l_{2i} = 0$. Thus equivalence clauses and XOR constraints are essentially the same concept.

An effective strategy for dealing with XOR constraints is to first scan the CNF encoding of a formula for sets of clauses that represent XOR constraints, and to then handle these constraints differently than ordinary clauses [Li00, HvM04, SNC09, Che09, Soo10, LJN10, LJN12a, LJN12b, Bie12, HJ12, SM19, SGM20].

The naive CNF encoding of an XOR constraint with k literals consists of 2^{k-1} clauses, where each clause rules out exactly one assignment that would falsify the constraint. More specifically, the CNF encoding of an XOR constraint $l_1 \oplus \dots \oplus l_k = b$ consists of all k -sized clauses over l_1, \dots, l_k , where an even number of literals occurs negated if $b = 1$ and an odd number occurs negated if $b = 0$.

Example 25. The XOR constraint

$$l_1 \oplus l_2 \oplus l_3 = 1$$

is falsified by every assignment that satisfies an even number of the literals l_1, l_2, l_3 . Its CNF encoding thus rules out all such assignments, leading to the clauses $(l_1 \vee l_2 \vee l_3), (\bar{l}_1 \vee \bar{l}_2 \vee l_3), (l_1 \vee \bar{l}_2 \vee \bar{l}_3), (\bar{l}_1 \vee l_2 \vee \bar{l}_3)$.

A popular method for dealing with XOR constraints, which is used by several solvers, including CRYPTOMINISAT [SNC09, Soo10], LINGELING [Bie12], and MARCH [HvM04], is *Gaussian elimination*. Gaussian elimination is based on the observation that XOR constraints can be viewed as equations over the finite field $\text{GF}(2)$ (also denoted \mathbb{F}_2 or \mathbb{Z}_2), where the \oplus operation corresponds to addition modulo 2 (and the \wedge operation corresponds to multiplication modulo 2).

A set of XOR constraints can then be seen as a system of linear equations, and we can derive new equations by adding together existing ones, just as known from high school algebra. If Gaussian elimination is performed in a preprocessing phase, it can possibly derive new unit clauses.

Example 26. Consider the clauses

$$(\bar{x} \vee y \vee z), (x \vee \bar{y} \vee z), (x \vee y \vee \bar{z}), (\bar{x} \vee \bar{y} \vee \bar{z}), \\ (y \vee z), (\bar{y} \vee \bar{z}).$$

These clauses correspond to the two XOR constraints

$$x \oplus y \oplus z = 0, \tag{9.3}$$

$$y \oplus z = 1. \tag{9.4}$$

As there are no unit clauses, unit propagation would not have an effect on the clauses. However, by adding the XOR constraint 9.3 to 9.4, and by using the fact that $l \oplus l = 0$ and $l \oplus 0 = l$ for every literal l , we can derive the XOR constraint

$$x = 1, \tag{9.5}$$

which is equivalent to the unit clause (x) .

Gaussian elimination can even show the unsatisfiability of particular formulas, as illustrated next.

Example 27. Consider the XOR constraints

$$x \oplus y = 1, \tag{9.6}$$

$$y \oplus z = 1, \tag{9.7}$$

$$x \oplus z = 1. \tag{9.8}$$

By adding 9.6 to 9.7, we derive

$$x \oplus z = 0. \tag{9.9}$$

If we add 9.8 to 9.9, we end up with the trivially unsatisfiable constraint $0 = 1$.

The use of Gaussian elimination can lead to exponential speedups in SAT solving. It has been shown that Tseitin formulas [Tse68] over expander graphs have only exponential-size resolution proofs [Urq87], meaning that CDCL solvers require exponential time to solve them. In contrast, Gaussian elimination can show the unsatisfiability of these formulas in polynomial time.

Moreover, Gaussian elimination can be expressed polynomially with extended resolution [SB06] and thus in the DRAT proof system [PR16]. It is still unclear though how many new variables are needed. For simple problems which require parity reasoning new variables can be avoided by reusing eliminated variables [CH20]. Note that Gaussian elimination can be seen as an instance of the more general approach of *algebraic* SAT solving in Chapter 7 on proof complexity.

There exist several approaches that extract and utilize equivalence clauses, especially in preprocessing. Already in 1998, Warners and van Maaren [WvM98] presented a preprocessor that utilizes linear programming to extract equivalence clauses from CNF formulas. By reasoning over these equivalence clauses in an initial solving phase and then running a basic DPLL procedure, their approach was the first to solve the notorious *parity formula* [CKS94] within reasonable time. Heule and van Maaren [HvM04] later improved this approach and integrated equivalence reasoning into the look-ahead solver MARCH (see Chapter 5 for details on look-ahead based solvers), which could achieve a considerable speedup by avoiding look-ups on variables for which look-ups on equivalent literals (as indicated via binary equivalences) had already been performed.

Another approach that relies on inference rules over equivalence clauses was presented by Li [Li00]. In an inprocessing fashion, he incorporated these inference rules—which apply to equivalence clauses with at most three literals—into the solver SATZ. The resulting solver, EQSATZ, was able to outperform the approach of Warners and van Maaren on some, but not all, of the parity formulas.

The current state of the art in parity reasoning, applied to approximate model counting, focuses on parity reasoning during search [SM19, SGM20], where much more efficient algorithms are required, extending ideas from [HJ12].

9.6.2. Circuit-Level Reasoning

Propositional encodings tend to be more natural to develop using the full language of propositional logic rather than aiming directly at a CNF-level encoding. Boolean circuits offer a succinct structural representation for propositional formulas. Succinctness comes from refining syntax trees of propositional formulas to directed acyclic graph structures by allowing *structural hashing*, i.e., by representing each subcircuit (subformula) only once. This is a common implementation technique for representing immutable data structures, also called *hash consing* in the context of functional languages; it is used in most BDD libraries (in form of a “unique table”) and related to *common subexpression elimination* in optimizing compilers. In practical implementations of circuit-level reasoning, restricted classes of circuits are used, with *and-inverter graphs* (AIGs) [KGP01] as one prominent example.

While SAT solvers working directly on the level of circuits have been developed (see, for instance, [Lar92] and in general Chapter 27), SAT solvers today tend to be implemented specifically for CNF formulas. In fact, standard Boolean constraint propagation achieved on the level of circuits is equivalently achieved by unit propagation on the CNF obtained via the Tseitin encoding [Tse68].

However, circuits offer a view to developing further structure-based CNF encodings and simplification techniques, by exploiting functional dependencies

explicit in circuits. A classical argument for the necessity of such structural information and against CNF is that this information is lost in the CNF encoding phase. While structural information is not evident in CNF, it has turned out that such arguments are partly wrong. In particular, CNF-level reasoning can achieve the same effects for various forms of circuit-level reasoning techniques.

One example is the Plaisted-Greenbaum CNF encoding of propositional formulas [PG86] which, based on the notion of polarities of subformulas, refines the Tseitin encoding [Tse68]. It allows to drop clauses representing one direction of the bi-implications used in the Tseitin encoding, thus locally encoding a unipolar subformula. While subformula polarities are computed following the circuit structure of a formula, it has been shown [JBH12] that blocked clause elimination, working “blindly” on the CNF, can remove all clauses left out by the Plaisted-Greenbaum encoding. This implies that blocked clause elimination from Section 9.4 achieves monotone-input reduction, i.e., it detects if the underlying circuit of the formula is monotone in terms of a particular input variable (input gate or primary input). As blocked clause elimination can be implemented efficiently, this renders the Plaisted-Greenbaum encoding on its own unnecessary.

Evidently, there are limits to what can be achieved (both in theory and in practice) in terms of structure-based simplifications solely on the CNF-level, but these limits are to an extent unclear. Structural hashing, for example, can be simulated by hyper binary resolution of Section 9.3.3 on the Tseitin encoding of AIGs, as shown in [HJB13]. However, for other gate types such as XORs and *if-then-else*, also called *multiplexer* (MUX), hyper binary resolution does not achieve structural hashing [HJB13].

More general hashing for minimizing circuit representations through detection of logically equivalent substructures can be achieved on the circuit level through BDD sweeping [KK97] and SAT sweeping [Kue04, MCJB05, ZKKS06, WKK12]. Stålmarck’s procedure [SS00] and Recursive Learning [KP92] are earlier circuit-level preprocessing techniques with similar effects.

There are CNF-level versions of Stålmarck’s procedure [GW00], Recursive Learning [MSG99] or more general SAT sweeping [HJB13, JLaMS15, CFM13]. These procedures allow to deduce literal equivalences implied by the CNF as well as literals forced to a specific truth value, because they have that same truth value in all models (so-called “backbones” [JLaMS15]). In contrast to hyper binary resolution, which also produces equivalences, sweeping is able to find more of these equivalences or even all, if required by some applications. These equivalences are typically used to reduce the CNF as described in Section 9.3.2.

The current state of the art in encoding circuits into CNF [EMS07] is based on techniques from circuit synthesis: *circuit-level rewriting* [BB04, BB06] with *cut enumeration* [MCB06a] and *technology mapping* [MCB06b]. Related work uses similar criteria [MV07, CMV09]. We are not aware of any CNF-level preprocessing technique able to simulate these optimizations.

When having to start with CNF, one approach to exploit circuit-level techniques is to first attempt to algorithmically recover a circuit-level representation potentially underlying the CNF formula, on which circuit-level techniques can then be applied. This process is also called *gate extraction* or *mining functional definitions*. Even though there is already a substantial amount of work

in this direction starting with [OGMS02, RMB04, FM07, IKS17] and most recently [LLM20, Sli20] with more references, the task of the recovery step is in general non-trivial and rarely used in practical SAT solving.

There is a related technique, called blocked clause decomposition [HB13, BFHB14, Che15], with a similar purpose: it partitions a given CNF into a set of blocked clauses (which is maximized) and some remaining clauses. The set of blocked clauses has similar properties as a circuit, i.e., clauses are ordered and models can be constructed polynomially, which has applications usually attributed to circuit-level reasoning, including SAT sweeping [HB13].

Beside circuits and XOR constraints already discussed in Section 9.6.1, there are other high-level representations or constraints for which dedicated preprocessing algorithms exist, but currently no (efficient) CNF-level technique is known. As explained for parity reasoning, a common approach is to extract such constraints, apply high-level preprocessing, and then encode the resulting constraints back to CNF. Closely related to XOR constraints are polynomials modulo 2 as considered in [CK07, CSCM19], and using Gröbner bases theory for preprocessing and inprocessing. Regarding pseudo-Boolean and cardinality constraints (see also Chapter 28) a similar argument applies [BLBLM14].

9.7. Conclusion

The development of novel types of preprocessing techniques and the optimized implementation of these techniques has made preprocessing a central part of the SAT solving workflow. In this chapter, we have discussed major developments in preprocessing in SAT, focusing mostly on CNF-level preprocessing techniques, but also considering some techniques on other representations.

In practice, the role of preprocessing is intertwined on one hand with the encoding phase, providing an automated way to re-encode CNF instances, and on the other hand with CDCL SAT solving. Developments in the latter direction have brought on the influential inprocessing SAT solving paradigm, which interleaves core CDCL search with complex combinations of preprocessing techniques.

Arguably, bounded variable elimination [EB05] is still the most important practical preprocessing technique today. However, the many other types of preprocessing techniques, developed in particular during the 21st century, play an evident role as well. In terms of supporting variable elimination, various other techniques are important for triggering further elimination steps of bounded variable elimination, by performing crucial redundancy-elimination steps that are not enabled by bounded variable elimination alone. Other individual techniques and their combinations can be central in different types of problem-specific applications of SAT solvers.

Going further, the various developments in CNF-level preprocessing have had a wide impact, well beyond mere preprocessing. By making use of polynomial-time checkable clause-redundancy properties, inprocessing SAT solving breaks the long-standing “resolution barrier” of standard CDCL SAT solvers, making the solvers more powerful than the resolution proof system. Clause redundancy properties have also resulted in generally applicable practical proof checking mechanisms, enabling the automated verification of proofs constructed by SAT solvers,

thus lifting the trustworthiness of “no” answers provided by SAT solvers to new levels (see Chapter 15).

There is much ongoing work on combining high-level and CNF-level preprocessing, particularly for non-linear algebraic reasoning [CSCM19, KBK19], while producing proofs for even parity reasoning is still a challenge [CH20]. In this context, revisiting techniques using binary decision diagrams might also be interesting [MM02, FKS⁺04, SB06, WFS06, BD09, vDEB18]. We also consider parallelization of preprocessing to still be in its infancy [WH13, GM13, HW13, BS18, OW19].

Finally, we want to point out that many of the preprocessing techniques discussed in this chapter have been lifted to more general logics and problems, such as QBF, MaxSAT, #SAT, and first-order logic, but instead of listing these extensions here, we refer to the chapters on these extensions in this handbook or the corresponding literature instead.

Acknowledgments

We would like to thank Lee Barnett, Katalin Fazekas, and Mathias Fleury for providing valuable feedback on late drafts of the chapter.

References

- [ABS99] G. Audemard, B. Benhamou, and P. Siegel. La méthode d’avalanche AVAL: Une méthode énumérative pour SAT. *Journées Nationales de la Résolution Pratique des problèmes NP-Complets (JNPC)*, pages 17–25, 1999.
- [ALS13] G. Audemard, J. Lagniez, and L. Simon. Improving Glucose for incremental SAT solving with assumptions: Application to MUS extraction. In M. Järvisalo and A. Van Gelder, editors, *SAT*, volume 7962 of *LNCIS*, pages 309–317. Springer, 2013.
- [APT79] B. Aspvall, M. F. Plass, and R. E. Tarjan. A linear-time algorithm for testing the truth of certain quantified Boolean formulas. *Inf. Process. Lett.*, 8(3):121–123, 1979.
- [AS09] G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In C. Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 399–404, 2009.
- [Bac02] F. Bacchus. Enhancing Davis Putnam with extended binary clause reasoning. In R. Dechter, M. J. Kearns, and R. S. Sutton, editors, *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada.*, pages 613–619. AAAI Press / The MIT Press, 2002.
- [BB04] P. Bjesse and A. Borälv. DAG-aware circuit compression for formal verification. In *2004 International Conference on Computer-Aided Design, ICCAD 2004, San Jose, CA, USA, November 7-11, 2004*, pages 42–49. IEEE Computer Society / ACM, 2004.

- [BB06] R. Brummayer and A. Biere. Local two-level And-Inverter graph minimization without blowup. In *Proceedings of the 2nd Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS 2006)*, 2006.
- [BBIS16] T. Balyo, A. Biere, M. Iser, and C. Sinz. SAT Race 2015. *Artificial Intelligence*, 241:45–65, 2016.
- [BCB20] L. A. Barnett, D. Cerna, and A. Biere. Covered clauses are not propagation redundant. In N. Peltier and V. Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I*, volume 12166 of *Lecture Notes in Computer Science*, pages 32–47. Springer, 2020.
- [BD09] M. Brickenstein and A. Dreyer. PolyBoRi: A framework for Gröbner-basis computations with Boolean polynomials. *J. Symb. Comput.*, 44(9):1326–1345, 2009.
- [BDP03] F. Bacchus, S. Dalmao, and T. Pitassi. DPLL with caching: A new algorithm for #SAT and Bayesian inference. *Electronic Colloquium on Computational Complexity (ECCC)*, 10(003), 2003.
- [BFFH20] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In T. Balyo, N. Froleyks, M. J. H. Heule, M. Iser, M. Järvisalo, and M. Suda, editors, *Proc. of SAT Competition 2020: Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.
- [BFHB14] T. Balyo, A. Fröhlich, M. J. H. Heule, and A. Biere. Everything you always wanted to know about blocked sets (but were afraid to ask). In C. Sinz and U. Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 317–332. Springer, 2014.
- [Bie03] A. Biere. About the SAT solvers Limmat, Compsat, Funex and the QBF solver Quantor, 2003. Presentation for the SAT’03 SAT Solver Competition.
- [Bie04] A. Biere. Resolve and expand. In H. H. Hoos and D. G. Mitchell, editors, *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers*, volume 3542 of *Lecture Notes in Computer Science*, pages 59–70. Springer, 2004.
- [Bie09] A. Biere. P{re,i}coSAT@SC’09. In *SAT 2009 Competitive Event Booklet*, 2009.
- [Bie10] A. Biere. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. Technical report, Johannes Kepler University Linz, 2010.
- [Bie11] A. Biere. Lingeling and friends at the SAT Competition 2011. Technical report, FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040

Linz, Austria, 2011.

- [Bie12] A. Biere. Lingeling and friends entering the SAT Challenge 2012. In *Proc. of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of *Dept. of Computer Science Series of Publications B*, University of Helsinki, 2012.
- [Bie14] A. Biere. Lingeling essentials, A tutorial on design and implementation aspects of the the SAT solver Lingeling. In D. Le Berre, editor, *POS-14. Fifth Pragmatics of SAT workshop, a workshop of the SAT 2014 conference, part of FLoC 2014 during the Vienna Summer of Logic, July 13, 2014, Vienna, Austria*, volume 27 of *EPiC Series in Computing*, page 88. EasyChair, 2014.
- [Bie16] A. Biere. Splatx, Lingeling, Plingeling, Treengeling, YalSAT entering the SAT Competition 2016. In T. Balyo, M. J. H. Heule, and M. Järvisalo, editors, *Proc. of SAT Competition 2016: Solver and Benchmark Descriptions*, volume B-2016-1 of *Department of Computer Science Series of Publications B*, pages 44–45. University of Helsinki, 2016.
- [Bie17] A. Biere. CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT entering the SAT Competition 2017. In T. Balyo, M. J. H. Heule, and M. Järvisalo, editors, *Proc. of SAT Competition 2017: Solver and Benchmark Descriptions*, volume B-2017-1 of *Department of Computer Science Series of Publications B*, pages 14–15. University of Helsinki, 2017.
- [Bie18] A. Biere. CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT entering the SAT Competition 2018. In M. J. H. Heule, M. Järvisalo, and M. Suda, editors, *Proc. of SAT Competition 2018: Solver and Benchmark Descriptions*, volume B-2018-1 of *Department of Computer Science Series of Publications B*, pages 13–14. University of Helsinki, 2018.
- [Bie19] A. Biere. CaDiCaL at the SAT Race 2019. In M. J. H. Heule, M. Järvisalo, and M. Suda, editors, *Proc. of SAT Race 2019: Solver and Benchmark Descriptions*, volume B-2019-1 of *Department of Computer Science Series of Publications B*, pages 8–9. University of Helsinki, 2019.
- [BKS04] P. Beame, H. A. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res.*, 22:319–351, 2004.
- [BLBLM14] A. Biere, D. Le Berre, E. Lonca, and N. Manthey. Detecting cardinality constraints in CNF. In C. Sinz and U. Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 285–301. Springer, 2014.
- [Blo70] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.
- [BP11] R. J. Bayardo and B. Panda. Fast algorithms for finding extremal

- sets. In *Proceedings of the Eleventh SIAM International Conference on Data Mining, SDM 2011, April 28-30, 2011, Mesa, Arizona, USA*, pages 25–34. SIAM / Omnipress, 2011.
- [Bra04] R. I. Brafman. A simplifier for propositional formulas with many binary clauses. *IEEE Trans. Systems, Man, and Cybernetics, Part B*, 34(1):52–59, 2004.
- [Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.
- [BS06] A. Biere and C. Sinz. Decomposing SAT problems into connected components. *J. Satisf. Boolean Model. Comput.*, 2(1-4):201–208, 2006.
- [BS18] T. Balyo and C. Sinz. Parallel satisfiability. In Y. Hamadi and L. Sais, editors, *Handbook of Parallel Constraint Reasoning*, pages 3–29. Springer, 2018.
- [BW03] F. Bacchus and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003, Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 341–355. Springer, 2003.
- [CFM13] M. Codish, Y. Fekete, and A. Metodi. Backbones for equality. In V. Bertacco and A. Legay, editors, *Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings*, volume 8244 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2013.
- [CH20] L. Chew and M. J. H. Heule. Sorting parity encodings by reusing variables. In L. Pulina and M. Seidl, editors, *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 2020.
- [Che09] J. Chen. Building a hybrid SAT solver via conflict-driven, look-ahead and XOR reasoning techniques. In O. Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 298–311. Springer, 2009.
- [Che15] J. Chen. Fast blocked clause decomposition with high quality. *CoRR*, abs/1507.00459, 2015.
- [CK07] C. Condrat and P. Kalla. A Gröbner basis approach to CNF-formulae preprocessing. In O. Grumberg and M. Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4424 of *Lecture Notes in Computer Science*, pages 618–631. Springer, 2007.

- [CKS94] J. M. Crawford, M. J. Kearns, and R. E. Schapire. The minimal disagreement parity problem as a hard satisfiability problem. Technical report, Computational Intelligence Research Laboratory and AT&T Bell Labs, 1994.
- [CMV09] B. Chambers, P. Manolios, and D. Vroon. Faster SAT solving with better CNF generation. In L. Benini, G. De Micheli, B. M. Al-Hashimi, and W. Müller, editors, *Design, Automation and Test in Europe, DATE 2009, Nice, France, April 20-24, 2009*, pages 1590–1595. IEEE, 2009.
- [Coo76] S. A. Cook. A short proof of the pigeon hole principle using extended resolution. *ACM SIGACT News*, 8(4):28–32, 1976.
- [CS00] P. Chatalic and L. Simon. ZRES: The old Davis-Putnam procedure meets ZBDD. In D. A. McAllester, editor, *Automated Deduction - CADE-17, 17th International Conference on Automated Deduction, Pittsburgh, PA, USA, June 17-20, 2000, Proceedings*, volume 1831 of *Lecture Notes in Computer Science*, pages 449–454. Springer, 2000.
- [CSCM19] D. Choo, M. Soos, K. M. A. Chai, and K. S. Meel. Bosphorus: Bridging ANF and CNF solvers. In J. Teich and F. Fummi, editors, *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*, pages 468–473. IEEE, 2019.
- [DFS59] B. Dunham, R. Fridshal, and G. L. Sward. A non-heuristic program for proving elementary logical theorems. In *Information Processing, Proceedings of the 1st International Conference on Information Processing, UNESCO, Paris 15-20 June 1959*, pages 282–284. UNESCO (Paris), 1959.
- [DLL62] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.
- [DM94] M. D’Agostino and M. Mondadori. The taming of the cut. classical refutations with analytic cut. *J. Log. Comput.*, 4(3):285–319, 1994.
- [DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.
- [dV00] A. del Val. On 2-SAT and renamable Horn. In H. A. Kautz and B. W. Porter, editors, *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence, July 30 - August 3, 2000, Austin, Texas, USA.*, pages 279–284. AAAI Press / The MIT Press, 2000.
- [dV01] A. del Val. Simplifying binary propositional theories into connected components twice as fast. In R. Nieuwenhuis and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, LPAR 2001, Havana, Cuba, December 3-7, 2001, Proceedings*, volume 2250 of *Lecture Notes in Computer Science*, pages 392–406. Springer, 2001.
- [EB05] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In F. Bacchus and T. Walsh, editors, *Theory and Applications of Satisfiability Testing, 8th International*

- Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.
- [EMS07] N. Eén, A. Mishchenko, and N. Sörensson. Applying logic synthesis for speeding up SAT. In J. a. Marques-Silva and K. A. Sakallah, editors, *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings*, volume 4501 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2007.
- [ES03] N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.
- [ES06] N. Eén and N. Sörensson. MiniSat 2.0 beta. In *Solver Description SAT Race 2006*, 2006.
- [FBS19] K. Fazekas, A. Biere, and C. Scholl. Incremental inprocessing in SAT solving. In M. Janota and I. Lynce, editors, *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, volume 11628 of *Lecture Notes in Computer Science*, pages 136–154. Springer, 2019.
- [FKS⁺04] J. V. Franco, M. Kouril, J. S. Schlipf, S. A. Weaver, M. R. Dransfield, and W. M. Vanfleet. Function-complete lookahead in support of efficient SAT search heuristics. *J. Univers. Comput. Sci.*, 10(12):1655–1695, 2004.
- [FM07] Z. Fu and S. Malik. Extracting logic circuit structure from conjunctive normal form descriptions. In *20th International Conference on VLSI Design (VLSI Design 2007), Sixth International Conference on Embedded Systems (ICES 2007), 6-10 January 2007, Bangalore, India*, pages 37–42. IEEE Computer Society, 2007.
- [Fre95] J. W. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, University of Pennsylvania, PA, USA, 1995.
- [GM13] K. Gebhardt and N. Manthey. Parallel variable elimination on CNF formulas. In I. J. Timm and M. Thimm, editors, *KI 2013: Advances in Artificial Intelligence - 36th Annual German Conference on AI, Koblenz, Germany, September 16-20, 2013. Proceedings*, volume 8077 of *Lecture Notes in Computer Science*, pages 61–73. Springer, 2013.
- [GS05] R. Gershman and O. Strichman. Cost-effective hyper-resolution for preprocessing CNF formulas. In F. Bacchus and T. Walsh, editors, *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*, pages 423–429. Springer, 2005.
- [GW00] J. F. Groote and J. P. Warners. The propositional formula checker HeerHugo. *J. Autom. Reasoning*, 24(1/2):101–125, 2000.
- [Hak85] A. Haken. The intractability of resolution. *Theor. Comput. Sci.*, 39:297–308, 1985.

- [HB13] M. J. H. Heule and A. Biere. Blocked clause decomposition. In K. L. McMillan, A. Middeldorp, and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, volume 8312 of *Lecture Notes in Computer Science*, pages 423–438. Springer, 2013.
- [HHJW13] M. J. H. Heule, W. A. Hunt Jr., and N. Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 181–188. IEEE, 2013.
- [HJ12] C. Han and J. R. Jiang. When Boolean satisfiability meets Gaussian elimination in a Simplex way. In P. Madhusudan and S. A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 410–426. Springer, 2012.
- [HJB10a] M. J. H. Heule, M. Järvisalo, and A. Biere. Clause elimination procedures for CNF formulas. In C. G. Fermüller and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, volume 6397 of *Lecture Notes in Computer Science*, pages 357–371. Springer, 2010.
- [HJB10b] M. J. H. Heule, M. Järvisalo, and A. Biere. Covered clause elimination. In A. Voronkov, G. Sutcliffe, M. Baaz, and C. G. Fermüller, editors, *Short papers for 17th International Conference on Logic for Programming, Artificial intelligence, and Reasoning, LPAR-17-short, Yogyakarta, Indonesia, October 10-15, 2010*, volume 13 of *EPiC Series in Computing*, pages 41–46. EasyChair, 2010.
- [HJB11] M. J. H. Heule, M. Järvisalo, and A. Biere. Efficient CNF simplification based on binary implication graphs. In K. A. Sakallah and L. Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, volume 6695 of *Lecture Notes in Computer Science*, pages 201–215. Springer, 2011.
- [HJB13] M. J. H. Heule, M. Järvisalo, and A. Biere. Revisiting hyper binary resolution. In C. P. Gomes and M. Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings*, volume 7874 of *Lecture Notes in Computer Science*, pages 77–93. Springer, 2013.
- [HJL⁺15] M. J. H. Heule, M. Järvisalo, F. Lonsing, M. Seidl, and A. Biere. Clause elimination for SAT and QSAT. *J. Artif. Intell. Res.*, 53:127–168, 2015.
- [HJN10] A. E. J. Hyvärinen, T. A. Junttila, and I. Niemelä. Partitioning SAT instances for distributed solving. In C. G. Fermüller and A. Voronkov, editors, *Logic for Programming, Artificial Intel-*

- ligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, volume 6397 of *Lecture Notes in Computer Science*, pages 372–386. Springer, 2010.
- [HJS10] Y. Hamadi, S. Jabbour, and L. Sais. Learning for dynamic subsumption. *Int. J. Artif. Intell. Tools*, 19(4):511–529, 2010.
- [HJS11] H. Han, H. Jin, and F. Somenzi. Clause simplification through dominator analysis. In *Design, Automation and Test in Europe, DATE 2011, Grenoble, France, March 14-18, 2011*, pages 143–148. IEEE, 2011.
- [HKB17] M. J. H. Heule, B. Kiesl, and A. Biere. Short proofs without new variables. In L. de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 130–147. Springer, 2017.
- [HKB20] M. J. H. Heule, B. Kiesl, and A. Biere. Strong extension-free proof systems. *J. Autom. Reason.*, 64(3):533–554, 2020.
- [Hoo93] J. N. Hooker. Solving the incremental satisfiability problem. *J. Log. Program.*, 15(1&2):177–186, 1993.
- [HS07] H. Han and F. Somenzi. Alembic: An efficient algorithm for CNF preprocessing. In *Proceedings of the 44th Design Automation Conference, DAC 2007, San Diego, CA, USA, June 4-8, 2007*, pages 582–587. IEEE, 2007.
- [HS09] H. Han and F. Somenzi. On-the-fly clause improvement. In O. Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 209–222. Springer, 2009.
- [HvM04] M. J. H. Heule and H. van Maaren. Aligning CNF- and equivalence-reasoning. In H. H. Hoos and D. G. Mitchell, editors, *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers*, volume 3542 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 2004.
- [HW13] Y. Hamadi and C. M. Wintersteiger. Seven challenges in parallel SAT solving. *AI Mag.*, 34(2):99–106, 2013.
- [IKS17] M. Iser, F. Kutzner, and C. Sinz. Using gate recognition and random simulation for under-approximation and optimized branching in SAT solvers. In *29th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2017, Boston, MA, USA, November 6-8, 2017*, pages 1029–1036. IEEE Computer Society, 2017.
- [JB10] M. Järvisalo and A. Biere. Reconstructing solutions after blocked clause elimination. In O. Strichman and S. Szeider, editors, *Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6175 of *Lecture Notes in Computer Science*, pages 340–345. Springer, 2010.

- [JBH10] M. Järvisalo, A. Biere, and M. J. H. Heule. Blocked clause elimination. In J. Esparza and R. Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*, pages 129–144. Springer, 2010.
- [JBH12] M. Järvisalo, A. Biere, and M. J. H. Heule. Simulating circuit-level simplifications on CNF. *J. Autom. Reasoning*, 49(4):583–619, 2012.
- [JHB12] M. Järvisalo, M. J. H. Heule, and A. Biere. Inprocessing rules. In B. Gramlich, D. Miller, and U. Sattler, editors, *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, volume 7364 of *Lecture Notes in Computer Science*, pages 355–370. Springer, 2012.
- [JK14] M. Järvisalo and J. H. Korhonen. Conditional lower bounds for failed literals and related techniques. In C. Sinz and U. Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 75–84. Springer, 2014.
- [JLaMS15] M. Janota, I. Lynce, and J. ao Marques-Silva. Algorithms for computing backbones of propositional formulae. *AI Commun.*, 28(2):161–177, 2015.
- [KBK19] D. Kaufmann, A. Biere, and M. Kauers. Verifying large multipliers by combining SAT and computer algebra. In C. W. Barrett and J. Yang, editors, *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019*, pages 28–36. IEEE, 2019.
- [KGP01] A. Kuehlmann, M. K. Ganai, and V. Paruthi. Circuit-based Boolean reasoning. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 232–237. ACM, 2001.
- [KHB19] B. Kiesl, M. J. H. Heule, and A. Biere. Truth assignments as conditional autarkies. In Y. Chen, C. Cheng, and J. Esparza, editors, *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings*, volume 11781 of *Lecture Notes in Computer Science*, pages 48–64. Springer, 2019.
- [KK97] A. Kuehlmann and F. Krohm. Equivalence checking using cuts and heaps. In E. J. Yoffa, G. De Micheli, and J. M. Rabaey, editors, *Proceedings of the 34th Conference on Design Automation, Anaheim, California, USA, Anaheim Convention Center, June 9-13, 1997.*, pages 263–268. ACM Press, 1997.
- [KN11] Z. Khasidashvili and A. Nadel. Implicative simultaneous satisfiability and applications. In K. Eder, J. ao Lourenço, and O. Shehory, editors, *Hardware and Software: Verification and Testing - 7th In-*

- ternational Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers, volume 7261 of *Lecture Notes in Computer Science*, pages 66–79. Springer, 2011.
- [KNPH05] Z. Khasidashvili, A. Nadel, A. Palti, and Z. Hanna. Simultaneous SAT-based model checking of safety properties. In S. Ur, E. Bin, and Y. Wolfsthal, editors, *Hardware and Software Verification and Testing, First International Haifa Verification Conference, Haifa, Israel, November 13-16, 2005, Revised Selected Papers*, volume 3875 of *Lecture Notes in Computer Science*, pages 56–75. Springer, 2005.
- [Knu15] D. E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Addison-Wesley Professional, 2015.
- [Kor08] K. Korovin. iProver — an instantiation-based theorem prover for first-order logic (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *Lecture Notes in Computer Science*, pages 292–298. Springer, 2008.
- [KP92] W. Kunz and D. K. Pradhan. Recursive learning: An attractive alternative to the decision tree for test generation in digital circuits. In *Proceedings IEEE International Test Conference 1992, Discover the New World of Test and Design, Baltimore, Maryland, USA, September 20-24, 1992*, pages 816–825. IEEE Computer Society, 1992.
- [KSS⁺17] B. Kiesl, M. Suda, M. Seidl, H. Tompits, and A. Biere. Blocked clauses in first-order logic. In T. Eiter and D. Sands, editors, *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*, volume 46 of *EPiC Series in Computing*, pages 31–48. EasyChair, 2017.
- [KSTB16] B. Kiesl, M. Seidl, H. Tompits, and A. Biere. Super-blocked clauses. In N. Olivetti and A. Tiwari, editors, *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, volume 9706 of *Lecture Notes in Computer Science*, pages 45–61. Springer, 2016.
- [KSTB18] B. Kiesl, M. Seidl, H. Tompits, and A. Biere. Local redundancy in SAT: Generalizations of blocked clauses. *Log. Methods Comput. Sci.*, 14(4), 2018.
- [Kue04] A. Kuehlmann. Dynamic transition relation simplification for bounded property checking. In *2004 International Conference on Computer-Aided Design, ICCAD 2004, San Jose, CA, USA, November 7-11, 2004*, pages 50–57. IEEE Computer Society / ACM, 2004.
- [Kul99] O. Kullmann. On a generalization of extended resolution. *Discret. Appl. Math.*, 96-97:149–176, 1999.
- [KWS00] J. Kim, J. Whittemore, and K. A. Sakallah. On solving stack-based incremental satisfiability problems. In *Proceedings of the IEEE International Conference On Computer Design: VLSI In Computers & Processors, ICCD '00, Austin, Texas, USA, September 17-20, 2000*, pages 379–382. IEEE Computer Society, 2000.

- [LaPMS03] I. Lynce and J. ao P. Marques-Silva. Probing-based preprocessing techniques for propositional satisfiability. In *15th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2003), 3-5 November 2003, Sacramento, California, USA*, pages 105–110. IEEE Computer Society, 2003.
- [Lar92] T. Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 11(1):4–15, 1992.
- [LB01] D. Le Berre. Exploiting the real power of unit propagation lookahead. *Electron. Notes Discret. Math.*, 9:59–80, 2001.
- [LE18] F. Lonsing and U. Egly. QRAT+: Generalizing QRAT by a more powerful QBF redundancy property. In D. Galniche, S. Schulz, and R. Sebastiani, editors, *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*, volume 10900 of *Lecture Notes in Computer Science*, pages 161–177. Springer, 2018.
- [LGPC16] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki. Learning rate based branching heuristic for SAT solvers. In N. Creignou and D. Le Berre, editors, *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 123–140. Springer, 2016.
- [Li00] C. M. Li. Integrating equivalency reasoning into Davis-Putnam procedure. In H. A. Kautz and B. W. Porter, editors, *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence, July 30 - August 3, 2000, Austin, Texas, USA.*, pages 291–296. AAAI Press / The MIT Press, 2000.
- [LJN10] T. Laitinen, T. A. Junttila, and I. Niemelä. Extending clause learning DPLL with parity reasoning. In H. Coelho, R. Studer, and M. Wooldridge, editors, *ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 21–26. IOS Press, 2010.
- [LJN12a] T. Laitinen, T. A. Junttila, and I. Niemelä. Classifying and propagating parity constraints. In M. Milano, editor, *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, volume 7514 of *Lecture Notes in Computer Science*, pages 357–372. Springer, 2012.
- [LJN12b] T. Laitinen, T. A. Junttila, and I. Niemelä. Extending clause learning SAT solvers with complete parity reasoning. In *IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012, Athens, Greece, November 7-9, 2012*, pages 65–72. IEEE Computer Society, 2012.
- [LLM20] J. Lagniez, E. Lonca, and P. Marquis. Definability for model count-

- ing. *Artif. Intell.*, 281:103229, 2020.
- [LLX⁺17] M. Luo, C. Li, F. Xiao, F. Manyà, and Z. Lü. An effective learnt clause minimization approach for CDCL SAT solvers. In C. Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 703–711. ijcai.org, 2017.
- [LXL⁺20] C. Li, F. Xiao, M. Luo, F. Manyà, Z. Lü, and Y. Li. Clause vivification by unit propagation in CDCL SAT solvers. *Artif. Intell.*, 279, 2020.
- [Man12] N. Manthey. Coprocessor 2.0 - A flexible CNF simplifier - (tool presentation). In A. Cimatti and R. Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 436–441. Springer, 2012.
- [MCB06a] A. Mishchenko, S. Chatterjee, and R. K. Brayton. DAG-aware AIG rewriting a fresh look at combinational logic synthesis. In E. Sentovich, editor, *Proceedings of the 43rd Design Automation Conference, DAC 2006, San Francisco, CA, USA, July 24-28, 2006*, pages 532–535. ACM, 2006.
- [MCB06b] A. Mishchenko, S. Chatterjee, and R. K. Brayton. Improvements to technology mapping for LUT-based FPGAs. In S. J. E. Wilton and A. DeHon, editors, *Proceedings of the ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays, FPGA 2006, Monterey, California, USA, February 22-24, 2006*, pages 41–49. ACM, 2006.
- [MCJB05] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton. FRAIGs: A unifying representation for logic synthesis and verification. Technical report, ERL Technical Report, 2005.
- [MHB12] N. Manthey, M. J. H. Heule, and A. Biere. Automated reencoding of Boolean formulas. In A. Biere, A. Nahir, and T. E. J. Vos, editors, *Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers*, volume 7857 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2012.
- [Min92] S. Minato. Fast generation of irredundant sum-of-products forms from binary decision diagrams. In *Proceedings of the Synthesis and Simulation Meeting and International Interchange (SASIMI'92)*, pages 64–73, 1992.
- [Min93] S. Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In *DAC*, pages 272–277. ACM Press, 1993.
- [MM02] D. B. Motter and I. L. Markov. A compressed breadth-first search for satisfiability. In D. M. Mount and C. Stein, editors, *Algorithm Engineering and Experiments, 4th International Workshop, ALENEX 2002, San Francisco, CA, USA, January 4-5, 2002, Revised Papers*, volume 2409 of *Lecture Notes in Computer Science*, pages 29–42. Springer, 2002.

- [MMZ⁺01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001.
- [MSG99] J. a. P. Marques-Silva and T. Glass. Combinational equivalence checking using satisfiability and recursive learning. In *1999 Design, Automation and Test in Europe (DATE '99), 9-12 March 1999, Munich, Germany*, pages 145–149. IEEE Computer Society / ACM, 1999.
- [MV07] P. Manolios and D. Vroon. Efficient circuit to CNF conversion. In J. ao Marques-Silva and K. A. Sakallah, editors, *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings*, volume 4501 of *Lecture Notes in Computer Science*, pages 4–9. Springer, 2007.
- [NII15] H. Nabeshima, K. Iwanuma, and K. Inoue. GlueMiniSat 2.2.10 & 2.2.10-5, 2015. SAT-Race 2015.
- [NR12] A. Nadel and V. Ryvchin. Efficient SAT solving under assumptions. In A. Cimatti and R. Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 242–255. Springer, 2012.
- [NR18] A. Nadel and V. Ryvchin. Chronological backtracking. In O. Beyersdorff and C. M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10929 of *Lecture Notes in Computer Science*, pages 111–121. Springer, 2018.
- [NRS12] A. Nadel, V. Ryvchin, and O. Strichman. Preprocessing in incremental SAT. In A. Cimatti and R. Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 256–269. Springer, 2012.
- [NRS14] A. Nadel, V. Ryvchin, and O. Strichman. Ultimately incremental SAT. In C. Sinz and U. Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 206–218. Springer, 2014.
- [OGMS02] R. Ostrowski, É. Grégoire, B. Mazure, and L. Sais. Recovering and exploiting structural knowledge from CNF formulas. In P. V. Hentenryck, editor, *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*, volume 2470 of *Lecture Notes in Computer Science*, pages 185–199. Springer, 2002.
- [Oh16] C. Oh. *Improving SAT solvers by exploiting empirical characteristics of CDCL*. PhD thesis, New York University, NY, USA, 2016.

- [OW19] M. Osama and A. Wijs. Parallel SAT simplification on GPU architectures. In *TACAS (1)*, volume 11427 of *Lecture Notes in Computer Science*, pages 21–40. Springer, 2019.
- [PD07] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In J. ao Marques-Silva and K. A. Sakallah, editors, *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer, 2007.
- [PD11] K. Pipatsrisawat and A. Darwiche. On the power of clause-learning SAT solvers as resolution engines. *Artif. Intell.*, 175(2):512–525, 2011.
- [PG86] D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *J. Symb. Comput.*, 2(3):293–304, 1986.
- [PHS08] C. Piette, Y. Hamadi, and L. Sais. Vivifying propositional clausal formulae. In M. Ghallab, C. D. Spyropoulos, N. Fakotakis, and N. M. Avouris, editors, *ECAI 2008 - 18th European Conference on Artificial Intelligence, Patras, Greece, July 21-25, 2008, Proceedings*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 525–529. IOS Press, 2008.
- [PR16] T. Philipp and A. Rebola-Pardo. DRAT proofs for XOR reasoning. In L. Michael and A. C. Kakas, editors, *Logics in Artificial Intelligence - 15th European Conference, JELIA 2016, Larnaca, Cyprus, November 9-11, 2016, Proceedings*, volume 10021 of *Lecture Notes in Computer Science*, pages 415–429, 2016.
- [RMB04] J. A. Roy, I. L. Markov, and V. Bertacco. Restoring circuit structure from SAT instances. In *Proceedings of International Workshop on Logic and Synthesis (IWLS)*, pages 663–678, 2004.
- [Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.
- [Rob74] J. A. Robinson. Automatic deduction with hyper-resolution. *J. Symb. Log.*, 39(1):189–190, 1974.
- [SAFS95] J. S. Schlipf, F. S. Annexstein, J. V. Franco, and R. P. Swaminathan. On finding solutions for extended Horn formulas. *Inf. Process. Lett.*, 54(3):133–137, 1995.
- [SB06] C. Sinz and A. Biere. Extended resolution proofs for conjoining BDDs. In D. Grigoriev, J. Harrison, and E. A. Hirsch, editors, *Computer Science - Theory and Applications, First International Computer Science Symposium in Russia, CSR 2006, St. Petersburg, Russia, June 8-12, 2006, Proceedings*, volume 3967 of *Lecture Notes in Computer Science*, pages 600–611. Springer, 2006.
- [SB09] N. Sörensson and A. Biere. Minimizing learned clauses. In O. Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 237–243. Springer, 2009.
- [SBB⁺04] T. Sang, F. Bacchus, P. Beame, H. A. Kautz, and T. Pitassi. Com-

- binning component caching and clause learning for effective model counting. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings, 2004*.
- [SCD⁺20] M. Soos, S. Cai, J. Devriendt, S. Gocht, A. Shaw, and K. Meel. CryptoMiniSAT with CCA_{nr} at the SAT Competition 2020. In T. Balyo, N. Froykys, M. J. H. Heule, M. Iser, M. Jarvisalo, and M. Suda, editors, *Proc. of SAT Competition 2020: Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 27–28. University of Helsinki, 2020.
- [SGM20] M. Soos, S. Gocht, and K. S. Meel. Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling. In S. K. Lahiri and C. Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 463–484. Springer, 2020.
- [Sli20] F. Slivovsky. Interpolation-based semantic gate extraction and its applications to QBF preprocessing. In S. K. Lahiri and C. Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 508–528. Springer, 2020.
- [SM19] M. Soos and K. S. Meel. BIRD: Engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 1592–1599. AAAI Press, 2019.
- [SNC09] M. Soos, K. Nohl, and C. Castelluccia. Extending SAT solvers to cryptographic problems. In O. Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009.
- [SNS02] P. Simons, I. Niemelä, and T. Soinen. Extending and implementing the stable model semantics. *Artif. Intell.*, 138(1-2):181–234, 2002.
- [Soo10] M. Soos. Enhanced Gaussian elimination in DPLL-based SAT solvers. In D. Le Berre, editor, *POS-10. Pragmatics of SAT, Edinburgh, UK, July 10, 2010*, volume 8 of *EPiC Series in Computing*, pages 2–14. EasyChair, 2010.
- [SP04] S. Subbarayan and D. K. Pradhan. NiVER: Non increasing variable elimination resolution for preprocessing SAT instances. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings, 2004*.

- [SS00] M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. *Formal Methods Syst. Des.*, 16(1):23–58, 2000.
- [SSK⁺20] M. Soos, B. Selman, H. Kautz, J. Devriendt, and S. Gocht. Crypto-MiniSAT with WalkSAT at the SAT Competition 2020. In T. Balyo, N. Froyleyks, M. J. H. Heule, M. Iser, M. Jarvisalo, and M. Suda, editors, *Proc. of SAT Competition 2020: Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 29–30. University of Helsinki, 2020.
- [Tar79] R. E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *J. Comput. Syst. Sci.*, 18(2):110–127, 1979.
- [Tse68] G. S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Mathematics and Mathematical Logic*, 2:115–125, 1968.
- [Urq87] A. Urquhart. Hard examples for resolution. *J. ACM*, 34(1):209–219, 1987.
- [vDEB18] T. van Dijk, R. Ehlers, and A. Biere. Revisiting decision diagrams for SAT. *CoRR*, abs/1805.03496, 2018.
- [VG05] A. Van Gelder. Toward leaner binary-clause reasoning in a satisfiability solver. *Ann. Math. Artif. Intell.*, 43(1):239–253, 2005.
- [VG12] A. Van Gelder. Producing and verifying extremely large propositional refutations - have your cake and eat it too. *Ann. Math. Artif. Intell.*, 65(4):329–372, 2012.
- [VGT93] A. Van Gelder and Y. K. Tsuji. Satisfiability testing with more reasoning and less guessing. In D. S. Johnson and M. A. Trick, editors, *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 11-13, 1993*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 559–586. DIMACS/AMS, 1993.
- [WFS06] S. A. Weaver, J. V. Franco, and J. S. Schlipf. Extending existential quantification in conjunctions of BDDs. *J. Satisf. Boolean Model. Comput.*, 1(2):89–110, 2006.
- [WH13] S. Wieringa and K. Heljanko. Concurrent clause strengthening. In M. Jarvisalo and A. Van Gelder, editors, *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, volume 7962 of *Lecture Notes in Computer Science*, pages 116–132. Springer, 2013.
- [WHHJ14] N. Wetzler, M. J. H. Heule, and W. A. Hunt Jr. Drat-trim: Efficient checking and trimming using expressive clausal proofs. In C. Sinz and U. Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, 2014.
- [WKK12] T. Welp, S. Krishnaswamy, and A. Kuehlmann. Generalized SAT-sweeping for post-mapping optimization. In P. Groeneveld, D. Sciuto, and S. Hassoun, editors, *The 49th Annual Design Automation*

Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012, pages 814–819. ACM, 2012.

- [WKS01] J. Whittimore, J. Kim, and K. A. Sakallah. SATIRE: A new incremental satisfiability engine. In *DAC*, pages 542–545. ACM, 2001.
- [WvdGS13] A. Wotzlaw, A. van der Grinten, and E. Speckenmeyer. Effectiveness of pre- and inprocessing for CDCL-based SAT solving. *CoRR*, abs/1310.4756, 2013.
- [WvM98] J. P. Warners and H. van Maaren. A two-phase algorithm for solving a class of hard satisfiability problems. *Oper. Res. Lett.*, 23(3-5):81–88, 1998.
- [ZC'20] X. Zhang and S. Cai. Relaxed backtracking with rephasing. In T. Balyo, N. Froleyks, M. J. H. Heule, M. Iser, M. Järvisalo, and M. Suda, editors, *Proc. of SAT Competition 2020: Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 15–16. University of Helsinki, 2020.
- [Zha05] L. Zhang. On subsumption removal and on-the-fly CNF simplification. In F. Bacchus and T. Walsh, editors, *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*, pages 482–489. Springer, 2005.
- [ZKKS06] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. L. Sangiovanni-Vincentelli. SAT sweeping with local observability don't-cares. In E. Sentovich, editor, *Proceedings of the 43rd Design Automation Conference, DAC 2006, San Francisco, CA, USA, July 24-28, 2006*, pages 229–234. ACM, 2006.
- [ZM88] R. Zabih and D. A. McAllester. A rearrangement search strategy for determining propositional satisfiability. In H. E. Shrobe, T. M. Mitchell, and R. G. Smith, editors, *Proceedings of the 7th National Conference on Artificial Intelligence, St. Paul, MN, USA, August 21-26, 1988.*, pages 155–160. AAAI Press / The MIT Press, 1988.
- [ZS00] H. Zhang and M. E. Stickel. Implementing the Davis-Putnam method. *J. Autom. Reason.*, 24(1/2):277–296, 2000.

This page intentionally left blank

Chapter 10

Random Satisfiability

Dimitris Achlioptas

10.1. Introduction

Satisfiability has received a great deal of study as the canonical NP-complete problem. In the last twenty years a significant amount of this effort has been devoted to the study of randomly generated satisfiability instances and the performance of different algorithms on them. Historically, the motivation for studying random instances has been the desire to understand the hardness of “typical” instances. In fact, some early results suggested that deciding satisfiability is “easy on average”. Unfortunately, while “easy” is easy to interpret, “on average” is not.

One of the earliest and most often quoted results for satisfiability being easy on average is due to Goldberg [Gol79]. In [FP83], though, Franco and Paull pointed out that the distribution of instances used in the analysis of [Gol79] is so greatly dominated by “very satisfiable” formulas that if one tries truth assignments completely at random, the expected number of trials until finding a satisfying one is $O(1)$. Alternatively, Franco and Paull pioneered the analysis of random instances of k -SAT, i.e., asking the satisfiability question for random k -CNF formulas (defined precisely below). Among other things, they showed [FP83] that for all $k \geq 3$ the DPLL algorithm needs an *exponential* number of steps to report all cylinders of solutions of such a formula, or that no solutions exist.

Random k -CNF formulas. Let $F_k(n, m)$ denote a Boolean formula in Conjunctive Normal Form with m clauses over n variables, where the clauses are chosen uniformly, independently and without replacement among all $2^k \binom{n}{k}$ non-trivial clauses of length k , i.e., clauses with k distinct, non-complementary literals.

Typically, k is a (small) fixed integer, while n is allowed to grow. In particular, we will say that a sequence of random events \mathcal{E}_n occurs with high probability (w.h.p.) if $\lim_{n \rightarrow \infty} \Pr[\mathcal{E}_n] = 1$ and with uniformly positive probability (w.u.p.p.) if $\liminf_{n \rightarrow \infty} \Pr[\mathcal{E}_n] > 0$. One of the first facts to be established [FP83] for random k -CNF formulas is that $F_k(n, rn)$ is w.h.p. unsatisfiable for $r \geq 2^k \ln 2$. A few years later, Chao and Franco [CF86, CF90] showed that, in contrast, if $r < 2^k/k$, a very simple algorithm will find a satisfying truth assignment w.u.p.p.

Combined, these two facts established $m = \Theta(n)$ as the most interesting regime for considering the satisfiability of random k -CNF formulas.

The study of random k -CNF formulas took off in the late 1980s. In a celebrated result, Chvátal and Szemerédi [CS88] proved that random k -CNF formulas w.h.p. have exponential resolution complexity, implying that if F is a random k -CNF formula with $r \geq 2^k \ln 2$, then w.h.p. every DPLL-type algorithm needs exponential time to prove its unsatisfiability. A few years later, Chvátal and Reed [CR92] proved that random k -CNF formulas are satisfiable w.h.p. for $r = O(2^k/k)$, strengthening the w.u.p.p. result of [CF90]. Arguably, the biggest boost came from the experimental work of Mitchell, Selman and Levesque [MSL92] and the analysis of these experiments by Kirkpatrick and Selman in [KS94].

In particular, [SML96] gave extensive experimental evidence suggesting that for $k \geq 3$, there is a range of the clauses-to-variables ratio, r , within which it seems hard even to *decide* if a randomly chosen k -SAT instance is satisfiable or not (as opposed to reporting all cylinders of solutions or that no solutions exist). The analysis of these experiments using finite-size scaling methods of statistical physics in [KS94] showed that this peak in experimental decision complexity coincided with a precipitous drop in the probability that a random formula is satisfiable. For example, for $k = 3$ the analysis drew the following remarkable picture: for $r < 4$, a satisfying truth assignment can be easily found for almost all formulas; for $r > 4.5$, almost all formulas are unsatisfiable; for $r \approx 4.2$, a satisfying truth assignment can be found for roughly half the formulas and around this point the computational effort for finding a satisfying truth assignment, whenever one exists, is maximized.

This potential connection between phase transitions and computational complexity generated a lot of excitement. A particularly crisp question is whether the probability of satisfiability indeed exhibits a sharp threshold around some critical density (ratio of clauses to variables).

Satisfiability Threshold Conjecture. *For every $k \geq 3$, there exists a constant $r_k > 0$ such that,*

$$\lim_{n \rightarrow \infty} \Pr[F_k(n, rn) \text{ is satisfiable}] = \begin{cases} 1, & \text{if } r < r_k \\ 0, & \text{if } r > r_k. \end{cases}$$

By now, the Satisfiability Threshold Conjecture has attracted attention in computer science, mathematics, and statistical physics. It has stimulated numerous interesting results and is at the center of an area of intense research activity, some of which we will survey in the following sections. As we will see, recent results have both strengthened the connection between computational complexity and phase transitions and shown that it is far more nuanced than originally thought. In particular, to the extent that finding satisfying assignments in random formulas is hard, this hardness comes from phase transitions in the solution-space geometry of the formulas, not in their probability of satisfiability. But this is getting ahead of our story.

To conclude this introduction we note that while a number of other generative models have been proposed for random SAT instances over the years (see [Fra01]

for an excellent survey) random k -SAT is by far the dominant model. One reason is that random k -CNF formulas enjoy a number of intriguing mathematical properties. Another is that random k -SAT instances remain computationally hard for a certain range of densities, making them a popular benchmark for testing and tuning satisfiability algorithms. In fact, some of the better practical ideas in use today come from insights gained by studying the performance of algorithms on random k -SAT instances [GSCK00].

The rest of this chapter is divided into two major pieces. The first piece is an overview of the current state of the art regarding mathematical properties of random k -CNF formulas where the number of clauses $m = \Theta(n)$. The second piece concerns the performance of different algorithms on sparse random k -CNF formulas. Both pieces are concerned with (and limited to) *rigorous mathematical results*. For experimental algorithmic results we refer the reader to the survey by Cook and Mitchell [CM97]. For results using the methods of statistical physics we refer the reader to Part 2, Chapter 22.

10.2. The State of the Art

10.2.1. The Satisfiability Threshold Conjecture

The satisfiability threshold conjecture remains open. A big step was made by Friedgut [Fri99] who showed the existence of a sharp threshold for satisfiability around a critical *sequence* of densities (rather than a single density r_k).

Theorem 1 ([Fri99]). *For every $k \geq 3$, there exists a sequence $r_k(n)$ such that for any $\epsilon > 0$,*

$$\lim_{n \rightarrow \infty} \Pr[F_k(n, m) \text{ is satisfiable}] = \begin{cases} 1, & \text{if } m = (r_k(n) - \epsilon)n \\ 0, & \text{if } m = (r_k(n) + \epsilon)n. \end{cases}$$

While it is widely believed that $r_k(n) \rightarrow r_k$, a proof remains elusive. A useful corollary of Friedgut's theorem is the following, as it allows one to give lower bounds for r_k by only establishing satisfiability w.u.p.p. rather than w.h.p.

Corollary 2. *If $F_k(n, r^*n)$ is satisfiable w.u.p.p., then for all $r < r^*$, $F_k(n, rn)$ is satisfiable w.h.p.*

Although the existence of r_k has not been established, we will allow ourselves the notational convenience of writing $r_k \leq r^*$ to denote that for $r > r^*$, $F_k(n, rn)$ is w.h.p. unsatisfiable (and analogously for $r_k \geq r^*$). In this notation, the current best bounds for the location of the satisfiability threshold are, roughly,

$$2^k \ln 2 - \Theta(k) \leq r_k \leq 2^k \ln 2 - \Theta(1) .$$

The precise form of these bounds is given by Theorem 3, which we illustrate for some small values of k in Table 10.1.

Theorem 3. *There exists sequences $\delta_k, \epsilon_k \rightarrow 0$ such that for all $k \geq 3$,*

$$2^k \ln 2 - (k+1) \frac{\ln 2}{2} - 1 - \delta_k \leq r_k \leq 2^k \ln 2 - \frac{1 + \ln 2}{2} + \epsilon_k .$$

It is very easy to show that $r_k \leq 2^k \ln 2$. This is because the probability that $F_k(n, m)$ is satisfied by at least one assignment is trivially bounded by 2^n times the probability that it is satisfied by any particular assignment, which, in turn, is bounded by $(1 - 2^{-k})^m$ (indeed, this last expression is exact if we assume that clauses are chosen with replacement). Since $2(1 - 2^{-k})^r < 1$ for $r \geq 2^k \ln 2$, this implies that for all such r the probability of satisfiability is exponentially small. The sharpening of this upper bound for general k is due, independently, to Dubois and Boufkhad [DB97] and Kirousis et al. [KKKS98]. It corresponds to bounding the probability of satisfiability by 2^n times the probability that a particular assignment is a *locally maximum* satisfying assignment, i.e., one in which no 0 can be turned into 1 without violating satisfiability (it is clear that every satisfiable formula has at least one such satisfying assignment, e.g., the lexicographically greatest one).

The lower bound in Theorem 3 is due to Achlioptas and Peres [AP04] and was proven via the, so-called, second moment method. It corresponds to the largest density for which $F_k(n, m)$ w.h.p. has *balanced* satisfying assignments, i.e., satisfying assignments in which the number of satisfied literals is $km/2 + O(n)$. In other words, balanced satisfying assignments have the property that in spite of being satisfying their number of satisfied literals is like that of a uniformly random $\sigma \in \{0, 1\}^n$. Focusing on balanced assignments is done of technical necessity for the second moment method to work (we discuss this point in detail in Section 10.6.3) and there does not appear to be an inherent reason for doing so. Indeed, as k grows, the upper bound of Theorem 3 coincides with the results¹ for the location of the threshold (up to a $o(1)$ term in k), giving even more evidence that the $O(k)$ term in the lower bound is an artifact of the analysis.

The second moment method, used to prove the lower bound, ignores individual solutions and captures, instead, statistical properties of the entire solution-space. As such, it offers no guidance whatsoever on how to efficiently *find* satisfying assignments for those densities for which it establishes their existence. Indeed, as we will see shortly, no efficient algorithm is known that can find solutions beyond density $O(2^k/k)$, even w.u.p.p. In the third row of Table 10.1 we indicate this phenomenon by giving the largest densities for which any efficient algorithm is known to succeed [KKL06, FS96]).

Finally, we note that while both general bounds above extend to $k = 3$, better bounds exist for that case. In particular, the lower bound for r_3 in Table 10.1 is actually algorithmic and due, independently, to Hajiaghayi and Sorkin [HS03] and Kaporis, Kirousis and Lalas [KKL06]. We discuss its derivation in Section 10.7.2. The upper bound is due to Dubois, Boufkhad and Mandler [DBM03] and uses the idea of locally maximum assignments mentioned above in combination with conditioning on the literal degree sequence, thus curtailing certain large deviations contributions that inflate the unconditional expectation.

¹We use the term *results* as a way of referring to “p[hysics] results” in this area, the subject of Part 2, Chapter 22. In general, results rest on combining mathematically rigorous arguments with highly non-trivial unproven assumptions, the latter often informed by general considerations of statistical physics. The term is also motivated by the fact that many results eventually became rigorous mathematical results via proofs that were deeply informed by the physical arguments.

Table 10.1. Best known rigorous bounds for the location of the satisfiability threshold for some small values of k . The last row gives the largest density for which a polynomial-time algorithm has been proven to find satisfying assignments.

k	3	4	5	7	10	20
Best upper bound9	,
Best lower bound	.	.9	.9	.	.9	, 9
Algorithmic lower bound	.	.	9.	.	.	9 ,

10.3. Random MAX k -SAT

The methods used to derive bounds for the location of the satisfiability threshold, also give bounds for the fraction of clauses that can be satisfied above it. Specifically, let us say that a k -CNF formula with m clauses is p -satisfiable, for some $p \in [0, 1]$, if there exists an assignment satisfying at least $(1 - \frac{1-p}{2^k})m$ clauses. Observe that every k -CNF formula is 0-satisfiable, since the average number of satisfied clauses over all assignments is $(1 - 2^{-k})m$, while satisfiability is simply 1-satisfiability. Thus, given $p \in (0, 1]$, the relevant quantity is the largest density, $r_k(p)$, for which a random k -CNF formula is w.h.p. p -satisfiable. Analogously to the case $p = 1$, the naive union bound over $\{0, 1\}^n$ for the existence of a p -satisfying truth assignment implies $r_k(p) \leq T_k(p)$, where

$$T_k(p) = \frac{2^k \ln 2}{p + (1-p) \ln(1-p)} .$$

Note that since $\lim_{p \rightarrow 1} T_k(p) = 2^k \ln 2$, this recovers the naive satisfiability upper bound. Using the second moment method, in [ANP07] it was shown that asymptotically, this upper bound is tight up to second order terms.

Theorem 4. *There exists a sequence $\delta_k = O(k2^{-k/2})$, such that for all $k \geq 2$ and $p \in (0, 1]$,*

$$(1 - \delta_k) T_k(p) \leq r_k(p) \leq T_k(p) .$$

Algorithms fall far short from this bound. Analogously to satisfiability, the best known algorithm [CGHS04] finds p -satisfying assignments only for $r = O(T_k(p)/k)$.

10.3.1. The case $k \in \{2, 3\}$

For $k = 2$, satisfiability can be decided in polynomial time using a very simple method: tentatively set any unset variable v to 0; repeatedly satisfy any 1-clauses that result; if a 0-clause is generated set v permanently to 1, otherwise set it permanently to 0. A 2-CNF formula is satisfiable iff when this process terminates no 0-clauses are present. For random 2-CNF formulas, the trees of implications generated by this process mimic the connected component structure of random digraphs. Using this connection, Chvátal and Reed [CR92], Goerdts [Goe96] and Fernandez de la Vega [FdIV92] independently proved $r_2 = 1$. Later, in [BBC⁺01], Bollobás et al. [BBC⁺01], also using this connection, determined the scaling window for random 2-SAT, showing that the transition from satisfiability to unsatisfiability occurs for $m = n + \lambda n^{2/3}$ as λ goes from $-\infty$ to $+\infty$.

In [MZK⁺, MZK⁺99a, MZ98, MZK⁺99b], Monasson et al., using mathematically sophisticated but non-rigorous techniques of statistical physics, initiated the analytical study of random CNF formulas that are mixtures of 2- and 3-clauses, which they dubbed $(2+p)$ -CNF. Such formulas arise for a number of reasons. For example, a frequent observation when converting problems from other domains into satisfiability problems is that they result into mixed CNF formulas with a substantial number of clauses of length 2, along with the clauses of length 3. Another reason is that DPLL algorithms run by recursively solving satisfiability on *residual formulas*, restricted versions of their input CNF formula, which are mixtures of clauses of length at least 2. When given random 3-CNF formulas as input, many DPLL algorithms produce residual formulas that are mixtures of random 2- and 3-clauses, making properties of random $(2+p)$ -CNF crucial for analyzing their running time.

A random $(2+p)$ -CNF on n variables with m clauses is formed by choosing pm clauses of length 3 and $(1-p)m$ clauses of length 2, amongst all clauses of each length, uniformly and independently. Thus, $p = 0$ corresponds to random 2-SAT, while $p = 1$ corresponds to random 3-SAT. Below we will find it convenient to sidestep this original formulation and state results directly in terms of the number of 2- and 3-clauses, not of the total number of clauses m and p .

The fact $r_2 = 1$ implies that for any $\epsilon > 0$ a random 2-CNF formula on n variables with $(1 - \epsilon/2)n$ clauses is w.h.p. satisfiable, but adding ϵn 2-clauses to it w.h.p. results in an unsatisfiable formula. The results in [MZK⁺99b] suggested, rather remarkably, that if instead of adding ϵn random 2-clauses one adds up to $0.703\dots n$ random 3-clauses, the formula remains satisfiable. In other words, that there is no finite “exchange rate” between 2- and 3-clauses.

Inspired by these claims, Achlioptas et al. [AKKK01] proved that

Theorem 5. *A random CNF formula with n variables, $(1 - \epsilon)n$ 2-clauses and Δn 3-clauses is w.h.p. satisfiable for all $\epsilon > 0$ and all $\Delta \leq 2/3$, but w.h.p. unsatisfiable for $\epsilon = 0.001$ and $\Delta = 2.28$.*

In other words, the physical prediction of an infinite exchange ratio is valid as one can add at least $0.66n$ 3-clauses (and no more than $2.28n$). In [Ach99] it was conjectured that the inequality $\Delta \leq 2/3$ in Theorem 5 is tight. That is,

Conjecture 6. *For all $\Delta > 2/3$, there exists $\epsilon = \epsilon(\Delta) > 0$, such that a random CNF formula with n variables, $(1 - \epsilon)n$ 2-clauses and Δn 3-clauses is w.h.p. unsatisfiable.*

Conjecture 6 is supported by a presult of Biroli, Monasson and Weigt [BMW00], subsequent to [MZK⁺, MZK⁺99a], asserting that $2/3$ is indeed tight. As we will see, if true, it implies that the running time of a large class of DPLL algorithms exhibits a sharp threshold behavior: for each algorithm \mathcal{A} , there exists a critical density $r_{\mathcal{A}} \leq r_k$ such that \mathcal{A} takes linear time for $r < r_{\mathcal{A}}$, but exponential time for $r > r_{\mathcal{A}}$.

10.3.2. Proof Complexity and its Implications for Algorithms

We saw that sparse random k -CNF formulas are hard to prove unsatisfiable using resolution [CS88]. Ben-Sasson and Impagliazzo [BSI99] and Alekhovich and

Razborov [AR01] proved that the same is true for the polynomial calculus, while Alekhovich [Ale05] proved its hardness for k -DNF resolution. Random k -CNF formulas are believed to be hard for other proof systems also, such as cutting planes, and this potential hardness has been linked to hardness of approximation [Fei02]. Moreover, the hardness of proving their unsatisfiability has been explored for dense formulas, where for sufficiently high densities it provably disappears [BKPS02, FGK05, GL03, COGLS04, COGL07]. We will focus on the resolution complexity of random k -CNF formulas as it has immediate and strong implications for the most commonly used satisfiability algorithm, namely the DPLL procedure.

10.3.2.1. Resolution Complexity of k -CNF formulas

The resolution rule allows one to derive a clause $(A \vee B)$ from two clauses $(A \vee x)$ and $(B \vee \bar{x})$. A resolution derivation of a clause C from a CNF formula F is a sequence of clauses $C_1, \dots, C_\ell = C$ such that each C_i is either a clause of F or follows from two clauses C_j, C_k for $j, k < i$ using the resolution rule. A resolution refutation of an unsatisfiable formula F is a resolution derivation of the empty clause. The size of a resolution refutation is its number of clauses.

In contrast, the Davis-Putnam/DLL (DPLL) algorithm on a CNF formula F performs a backtracking search for a satisfying assignment of F by extending partial assignments until they either reach a satisfying assignment or violate a clause of F . It is well known that for an unsatisfiable formula F , the tree of nodes explored by any DPLL algorithm can be converted to a resolution refutation of F where the pattern of inferences forms the same tree.

For random k -CNF formulas in the unsatisfiable regime, the behavior of DPLL algorithms, and the more general class of resolution-based algorithms, is well-understood. Specifically, since every unsatisfiable 2-CNF formula has a linear-size resolution refutation, if $r > 1$ then even the simplest DPLL algorithms w.h.p. run in polynomial time on a random 2-CNF formula. On the other hand, for $k \geq 3$ the aforementioned result of Chvátal and Szemerédi [CS88] asserts that w.h.p. a random k -CNF formula in the unsatisfiable regime requires an exponentially long resolution proof of unsatisfiability. More precisely, let $\text{res}(F)$ be the size of the minimal resolution refutation a formula F (assume $\text{res}(F) = \infty$ if F is satisfiable). In [CS88] it was proved that

Theorem 7. *For all $k \geq 3$ and any constant $r > 0$, w.h.p. $\text{res}(F_k(n, rn)) = 2^{\Omega(n)}$.*

Corollary 8. *Every DPLL algorithm w.h.p. takes exponential time on $F_k(n, rn)$, for any constant $r \geq 2^k \ln 2$.*

10.3.2.2. $(2 + p)$ -CNF formulas and their Algorithmic Implications

Since all unsatisfiable 2-CNF formulas have linear-size resolution refutations and $r_2 = 1$, it follows that adding $(1 + \epsilon)n$ random 2-clauses to a random 3-CNF formula w.h.p. causes its resolution complexity to collapse from exponential to linear. In [ABM04b], Achlioptas, Beame and Molloy proved that, in contrast, adding $(1 - \epsilon)n$ random 2-clauses w.h.p. has essentially no effect on its proof complexity.

Theorem 9. For any constants $r, \epsilon > 0$, let F be a random formula with n variables, $(1 - \epsilon)n$ 2-clauses and rn 3-clauses. *W.h.p.* $\text{res}(F) = 2^{\Omega(n)}$.

Theorem 9 allows one to readily prove exponential lower bounds for the running times of DPLL algorithms for *satisfiable* random k -CNF formulas. This is because many natural DPLL algorithms when applied to random k -CNF formulas generate at least one unsatisfiable subproblem consisting of a random mixture of 2- and higher-length clauses, where the 2-clauses alone are satisfiable. (We will discuss how, when and for which algorithms this happens in greater detail in Section 10.9.) By Theorem 9, such a mixture has exponential resolution complexity (converting k -clauses with $k > 3$ to 3-clauses arbitrarily can only reduce resolution complexity) and, as a result, to resolve any such subproblem (and backtrack) any DPLL algorithm needs exponential time.

10.4. Physical Predictions for Solution-space Geometry

Random k -SAT, along with other random Constraint Satisfaction Problems, such as random graph coloring and random XOR-SAT have also been studied systematically in physics in the past two decades. For a general introduction and exposition to the physical methods see Part 2, Chapter 22. In particular, motivated by ideas developed for the study of materials known as spin glasses, physicists have put forward a wonderfully complex picture about how the geometry of the set of satisfying assignments of a random k -CNF formula evolves as clauses are added. Perhaps the most detailed and sophisticated version of this picture comes from [KMRT⁺07].

Roughly speaking, statistical physicists have predicted (using non-rigorous but mathematically sophisticated methods) that while for low densities the set of satisfying assignments forms a single giant cluster, at some critical density this cluster shatters into exponentially many clusters, each of which is relatively tiny and far apart from all other clusters. These clusters are further predicted to be separated from one another by huge “energy barriers”, i.e., every path connecting satisfying assignments in different clusters must pass through assignments that violate $\Omega(n)$ constraints. Moreover, inside each cluster the majority of variables are predicted to be frozen, i.e., take the same value in all solutions in the cluster; thus getting even a single frozen variable wrong requires traveling $\Omega(n)$ away and over a huge energy barrier to correct it.

The regime of exponentially many tiny clusters, each one having constant probability of vanishing every time a clause is added (due to its frozen variables), is predicted in [KMRT⁺07] to persist until very close to the threshold, namely for densities up to

$$2^k \ln 2 - \frac{3 \ln 2}{2} + o(1) , \quad (10.1)$$

where the $o(1)$ term is asymptotic in k . In comparison, the satisfiability threshold is predicted [MMZ06] to occur at

$$2^k \ln 2 - \frac{1 + \ln 2}{2} + o(1) , \quad (10.2)$$

i.e., fewer than $0.2n$ k -clauses later. For densities between (10.1) and (10.2), it is predicted that nearly all satisfying assignments lie in a small (finite) number of (atypically large) clusters, while exponentially many clusters still exist.

10.4.1. The Algorithmic Barrier

Perhaps the most remarkable aspect of the picture put forward by physicists is that the predicted density for the shattering of the set of solutions into exponentially many clusters scales, asymptotically in k , as

$$\ln k \cdot \frac{2^k}{k}, \quad (10.3)$$

fitting perfectly with the fact that all known algorithms fail at some density

$$c_{\mathcal{A}} \cdot \frac{2^k}{k}, \quad (10.4)$$

where the constant $c_{\mathcal{A}}$ depends on the algorithm. We will refer to this phenomenon as the “algorithmic barrier”. We note that so far there does not appear to be some natural general upper bound for $c_{\mathcal{A}}$ for the types of algorithms that have been analyzed, i.e., more sophisticated algorithms do have a greater constant, but with rapidly diminishing returns in terms of their complexity.

10.4.2. Rigorous Results for Solution-space Geometry

By now a substantial part of the picture put forward by physicists has been made rigorous, at least for $k \geq 8$. Success for smaller k stumbles upon the fact that the rigorous results rely on the second moment method which does not seem to perform as well for small k .

For the definitions in the rest of this section we assume we are dealing with an arbitrary CNF formula F defined over variables $X = x_1, \dots, x_n$, and we let $\mathcal{S}(F) \subseteq \{0, 1\}^n$ denote its set of satisfying assignments. We say that two assignments are adjacent if their Hamming distance is 1 and we let $H_F : \{0, 1\}^n \rightarrow \mathbb{N}$ be the function counting the number of clauses of F violated by each $\sigma \in \{0, 1\}^n$.

Definition 10. The **clusters** of a formula F are the connected components of $\mathcal{S}(F)$. A **region** is a non-empty union of clusters. The **height** of any path $\sigma_0, \sigma_1, \dots, \sigma_t \in \{0, 1\}^n$ is $\max_i H(\sigma_i)$.

One can get an easy result regarding the solution-space geometry of *very* sparse random formulas by observing that if a formula F is satisfiable by the *pure literal* rule alone, then the set $\mathcal{S}(F)$ is connected (recall that the pure literal rule amounts to permanently satisfying any literal ℓ whose complement does not appear in the formula and permanently removing all clauses containing ℓ ; the proof of connectivity is left as an exercise). The pure literal rule alone w.h.p. finds satisfying assignments in random k -CNF formulas with up to $\rho_k \cdot n$ clauses, for some $\rho_k \rightarrow 0$, so this is very far away from the densities up to which the best known algorithms succeed, namely $O(2^k/k)$.

The following theorem of Achlioptas and Coja-Oghlan [ACO08], on the other hand, asserts that for all $k \geq 8$, precisely at the asymptotic density predicted by physics in (10.3), the set of satisfying assignments shatters into an exponential number of well-separated regions. Given two functions $f(n), g(n)$, let us write $f \sim g$ if $\lim_{n \rightarrow \infty} f(n)/g(n) = 1$. Recall that the lower bound for the location of the satisfiability threshold provided by the second moment method is $s_k \equiv 2^k \ln 2 - (k+1) \frac{\ln 2}{2} - 1 - o(1) \sim 2^k \ln 2$.

Theorem 11. *For all $k \geq 8$, there exists $c_k < s_k$ such that for all $r \in (c_k, s_k)$, the set $\mathcal{S}(F_k(n, rn))$ w.h.p. consists of exponentially many regions where:*

1. *Each region only contains an exponentially small fraction of \mathcal{S} .*
2. *The Hamming distance between any two regions is $\Omega(n)$.*
3. *Every path between assignments in distinct regions has height $\Omega(n)$.*

In particular, $c_k \sim \ln k \cdot 2^k/k$.

The picture of Theorem 11 comes in even sharper focus for large k . In particular, for sufficiently large k , sufficiently close to the threshold, the regions become arbitrarily small and maximally far apart, while still exponentially many.

Theorem 12. *For any $0 < \delta < 1/3$, fix $r = (1 - \delta)2^k \ln 2$, and let*

$$\alpha_k = \frac{1}{k}, \quad \beta_k = \frac{1}{2} - \frac{5}{6}\sqrt{\delta}, \quad \epsilon_k = \frac{\delta}{2} - 3k^{-2}.$$

For all $k \geq k_0(\delta)$, w.h.p. the set $\mathcal{S} = \mathcal{S}(F_k(n, rn))$ consists of $2^{\epsilon_k n}$ regions where:

1. *The diameter of each region is at most $\alpha_k n$.*
2. *The distance between every pair of regions is at least $\beta_k n$.*

Thus, the solution-space geometry becomes like that of an correcting code with a little bit of “fuzz” around each codeword. This “shattering” phase transition is known as a *dynamical* transition, while the transition in which nearly all assignments concentrate on a finite number of clusters is known as a *condensation* transition. This dynamical phase transition has strong negative repercussions for the performance of random-walk type algorithms on random k -CNF formulas and exploring them precisely is an active area of research.

Remark 13. *The presults of [KMRT⁺07] state that the picture of Theorem 11 should hold for all $k \geq 4$, but **not** for $k = 3$. In particular for $k = 3$, the solution-space geometry is predicted to pass from a single cluster to a condensed phase directly.*

It turns out that after the set of satisfying assignments has shattered into exponentially many clusters, we can “look inside” these clusters and make some statements about their shape. For that, we need the following definition.

Definition 14. *The **projection** of a variable x_i over a set of assignments C , denoted as $\pi_i(C)$, is the union of the values taken by x_i over the assignments in C . If $\pi_i(C) \neq \{0, 1\}$ we say that x_i is **frozen** in C .*

Theorem 15 below [ACO08] asserts that the dynamical transition is followed, essentially immediately, by the massive appearance of frozen variables.

Theorem 15. *For all $k \geq 8$, there exists $d_k < s_k$ such that for all $r \in (d_k, s_k)$, a random $\sigma \in \mathcal{S}(F_k(n, rn))$ w.h.p. has at least $\gamma_k \cdot n$ frozen variables, where $\gamma_k \rightarrow 1$ for all $r \in (d_k, s_k)$. In particular, $d_k \sim \ln k \cdot 2^k/k$.*

The first rigorous analysis of frozen variables by Achlioptas and Ricci-Tersenghi [ART06] also establishes that

Corollary 16. *For every $k \geq 9$, there exists $r < s_k$ such that w.h.p. every cluster of $F_k(n, rn)$ has frozen variables.*

It remains open whether frozen variables exist for $k < 8$.

10.5. The Role of the Second Moment Method

Recall that the best upper bound for the satisfiability threshold scales as $\Theta(2^k)$, whereas all efficient algorithms known work only for densities up to $O(2^k/k)$. To resolve whether this gap was due to a genuine lack of solutions, as opposed to the difficulty of finding them, Achlioptas and Moore [AM06] introduced an approach which allows one to avoid the pitfall of computational complexity: namely, using the second-moment method one can prove that solutions exist in random instances, without the need to identify any particular solution for each instance (as algorithms do). Indeed, if random formulas are genuinely hard at some densities below the satisfiability threshold, then focusing on the *existence* of solutions rather than their efficient discovery is essential: one cannot expect algorithms to provide accurate results on the threshold's location; they simply cannot get there!

Before we delve into the ideas underlying some of the results mentioned above it is helpful to establish the probabilistic equivalence between a few different models for generating random k -CNF formulas.

10.6. Generative models

Given a set V of n Boolean variables, let $C_k = C_k(V)$ denote the set of all proper k -clauses on V , i.e., the set of all $2^k \binom{n}{k}$ disjunctions of k literals involving distinct variables. As we saw, a random k -CNF formula $F_k(n, m)$ is formed by selecting a uniformly random m -subset of C_k . While $F_k(n, m)$ is perhaps the most natural model for generating random k -CNF formulas, there are a number of slight variations of the model, largely motivated by their amenability to calculations.

For example, it is fairly common to consider the clauses as ordered k -tuples (rather than as k -sets) and/or to allow replacement in sampling the set C_k . Clearly, for properties such as satisfiability the issue of ordering is irrelevant. Moreover, as long as $m = O(n)$, essentially the same is true for the issue of replacement. To see that, observe that w.h.p. the number, q of repeated clauses is $o(n)$, while the set of $m - q$ distinct clauses is a uniformly random $(m - q)$ -subset of C_k . Thus, if a monotone decreasing property (such as satisfiability) holds with

probability p for a given $m = r^*n$ when replacement is allowed, it holds with probability $p - o(1)$ for all $r < r^*$ when replacement is not allowed.

The issue of selecting the literals of each clause with replacement (which might result in some “improper” clauses) is completely analogous. That is, the probability that a variable appears more than once in a given clause is at most $k^2/n = O(1/n)$ and hence w.h.p. there are $o(n)$ improper clauses. Finally, we note that by standard techniques, e.g., see [FS96], results also transfer between $F_k(n, m)$ and the model where every clause in C_k is included in the formula independently of all others with probability p , when $pC_k \sim m$.

10.6.1. The Vanilla Second Moment Method

The second moment method approach [ANP05] rests on the following basic fact: every non-negative random variable X satisfies $\Pr[X > 0] \geq \mathbf{E}[X]^2 / \mathbf{E}[X^2]$. Given any k -SAT instance F on n variables, let $X = X(F)$ be its number of satisfying assignments. By computing $\mathbf{E}[X^2]$ and $\mathbf{E}[X]^2$ for random formulas with a given density r one can hope to get a lower bound on the probability that $X > 0$, i.e., that $F_k(n, rn)$ is satisfiable. Unfortunately, this direct application fails dramatically as $\mathbf{E}[X^2]$ is exponentially (in n) greater than $\mathbf{E}[X]^2$ for every density $r > 0$. Nevertheless, it is worth going through this computation below as it points to the source of the problem and also helps in establishing the existence of clusters. We perform all computations below in the model where clauses are chosen with replacement from C_k .

For a k -CNF formula with m clauses chosen independently with replacement it is straightforward to show that the number of satisfying assignments X satisfies

$$\mathbf{E}[X^2] = \sum_{z=0}^n 2^n \binom{n}{z} f_S(z/n)^m, \quad (10.5)$$

where $f_S(\alpha) = 1 - 2^{1-k} + 2^{-k} \alpha^k$ is the probability that two fixed truth assignments that agree on $z = \alpha n$ variables, *both* satisfy a randomly drawn clause. Thus, (10.5) decomposes the second moment of X into the expected number of pairs of satisfying assignments at each possible distance.

Observe that f is an increasing function of α and that $f_S(1/2) = (1 - 2^{-k})^2$, i.e., truth assignments at distance $n/2$ are uncorrelated. Using the approximation $\binom{n}{\alpha n} = (\alpha^\alpha (1-\alpha)^{1-\alpha})^{-n} \times \text{poly}(n)$ and letting

$$\Lambda_S(\alpha) = \frac{2 f_S(\alpha)^r}{\alpha^\alpha (1-\alpha)^{1-\alpha}}$$

we see that

$$\begin{aligned} \mathbf{E}[X^2] &= \left(\max_{0 \leq \alpha \leq 1} \Lambda_S(\alpha) \right)^n \times \text{poly}(n), \\ \mathbf{E}[X]^2 &= \Lambda_S(1/2)^n. \end{aligned}$$

Therefore, if there exists some $\alpha \neq 1/2$ such that $\Lambda_S(\alpha) > \Lambda_S(1/2)$, then the second moment is exponentially greater than the square of the expectation and we

only get an exponentially small lower bound for $\Pr[X > 0]$. Put differently, unless the dominant contribution to $\mathbf{E}[X^2]$ comes from uncorrelated pairs of satisfying assignments, i.e., pairs with overlap $n/2$, the second moment method fails.

Unfortunately, this is precisely what happens for all $r > 0$ since the entropic factor $\mathcal{E}(\alpha) = 1/(\alpha^\alpha(1-\alpha)^{1-\alpha})$ in Λ_S is symmetric around $\alpha = 1/2$, while f_S is increasing in $(0, 1)$, implying $\Lambda'_S(1/2) > 0$. That is, Λ_S is maximized at some $\alpha > 1/2$ where the correlation benefit balances the penalty of decreased entropy.

While the above calculation does not give us what we want, it is still useful. For example, for any real number $\alpha \in [0, 1]$, it would be nice to know the number of pairs of satisfying truth assignments that agree on $z = \alpha n$ variables in a random formula. Each term in the sum in (10.5) gives us the *expected* number of such pairs. While this expectation may overemphasize formulas with more satisfying assignments (as they contribute more heavily to the expectation), it still gives valuable information on the distribution of distances among truth assignments in a random formula. For example, if for some values of z (and k, r) this expectation tends to 0 with n , we can readily infer that w.h.p. there are no pairs of truth assignments that agree on z variables in $F_k(n, rn)$. This is because for any integer-valued random variable Y , $\Pr[Y > 0] \leq \mathbf{E}[Y]$. Indeed, this simple argument is enough to provide the existence of clustering for all $k \geq 8$, at densities in the $\Theta(2^k)$ range (getting down to $(2^k/k) \cdot \ln k$ requires a lot more work).

10.6.2. Proving the Existence of Exponentially Many Clusters

To prove the existence of exponentially many regions one divides a lower bound for the total number of satisfying assignments with an upper bound for the number of truth assignments in each region. The lower bound comes from the expected number of *balanced* satisfying assignments since the success of the second moment method for such assignments, which we will see immediately next, implies that w.h.p. the actual number of balanced assignments is not much lower than its expectation. For the upper bound, one bounds the total number of *pairs* of truth assignments in each region as $\text{poly}(n) \times g(k, r)^n$, where

$$g(k, r) = \max_{\alpha \in [0, \Delta]} \Lambda_S(\alpha, k, r) ,$$

where Δ is the smallest number such that $\Lambda_S(\Delta, k, r) < 1$, i.e., Δn is a bound on the diameter of any region. Dividing the lower bound for the total number of satisfying assignments with the square root of this quantity yields the existence of exponentially many clusters.

10.6.3. Weighted second moment: the importance of being balanced

An attractive feature of the second moment method is that we are free to apply it to any random variable $X = X(F)$ such that $X > 0$ implies that F is satisfiable. With this in mind, let us consider random variables of the form

$$X = \sum_{\sigma} \prod_c w(\sigma, c)$$

where w is some arbitrary function. (Eventually, we will require that $w(\sigma, c) = 0$ if σ falsifies c .) Similarly to (10.5), it is rather straightforward to prove that

$$\mathbf{E}[X^2] = 2^n \sum_{z=0}^n \binom{n}{z} f_w(z/n)^m ,$$

where $f_w(z/n) = \mathbf{E}[w(\sigma, c)w(\tau, c)]$ is the correlation, with respect to a single random clause c , between two truth assignments σ and τ that agree on z variables. It is also not hard to see that $f_w(1/2) = \mathbf{E}[w(\sigma, c)]^2$, i.e., truth assignments at distance $n/2$ are uncorrelated for *any* function w . Thus, arguing as in the previous section, we see that $\mathbf{E}[X^2]$ is exponentially greater than $\mathbf{E}[X]^2$ unless $f'_w(1/2) = 0$.

At this point we observe that since we are interested in random formulas where literals are drawn uniformly, it suffices to consider functions w such that: for every truth assignment σ and every clause $c = \ell_1 \vee \dots \vee \ell_k$, $w(\sigma, c) = w(\mathbf{v})$, where $v_i = +1$ if ℓ_i is satisfied under σ and -1 if ℓ_i is falsified under σ . (So, we will require that $w(-1, \dots, -1) = 0$.) Letting $A = \{-1, +1\}^k$ and differentiating f_w , yields the geometric condition

$$f'_w(1/2) = 0 \iff \sum_{\mathbf{v} \in A} w(\mathbf{v})\mathbf{v} = 0 . \quad (10.6)$$

The condition in the r.h.s. of (10.6) asserts that the vectors in A , when scaled by $w(\mathbf{v})$, must cancel out. This gives us another perspective on the failure of the vanilla second moment method: when $w = w_S$ is the indicator variable for satisfiability, the condition in the right hand side of (10.6) does not hold since the vector $(-1, -1, \dots, -1)$ has weight 0, while all other $\mathbf{v} \in A$ have weight 1.

Note that the r.h.s. of (10.6) implies that in a successful w each coordinate must have mean 0, i.e., that each literal must be equally likely to be $+1$ or -1 when we pick truth assignments with probability proportional to their weight under w . We will call truth assignments with $km/2 \pm O(\sqrt{m})$ satisfied literal occurrences *balanced*. As was shown in [AP04], if X is the number of balanced satisfying assignments then $\mathbf{E}[X^2] < C \cdot \mathbf{E}[X]^2$, for all $r \leq s_k$, where $C = C(k) > 0$ is independent of n . Thus, $\Pr[X > 0] \geq 1/C$ and by Corollary 2 we get $r_k \geq s_k$.

To gain some additional intuition on the success of balanced assignments it helps to think of $F_k(n, m)$ as generated in two steps: first choose the km literal occurrences randomly, and then partition them randomly into k -clauses. At the end of the first step, truth assignments that satisfy many literal occurrences clearly have significantly greater conditional probability of eventually being satisfying assignments. But such assignments are highly correlated with each other since in order to satisfy many literal occurrences they tend to agree with the majority truth assignment on more than half the variables. Focusing on balanced assignments only, curbs the tendency of satisfying assignments to lean towards the majority vote assignment.

Finally, we note that it is not hard to prove that for $r \geq s_k + O(1)$, w.h.p. $F_k(n, rn)$ has no satisfying truth assignments that only satisfy $km/2 + o(n)$ literal occurrences. Thus, any asymptotic improvement over the lower bound for r_k provided by balanced assignments would mean that tendencies toward the majority assignment become essential as we approach the threshold. By the same token,

we note that as k increases the influence exerted by the majority vote assignment becomes less and less significant as most literals occur very close to their expected $kr/2$ times. As a result, as k increases, typical satisfying assignments get closer and closer to being balanced, meaning that the structure of the space of solutions for small values of k (e.g., $k = 3, 4$) might be significantly different from the structure for large values of k , something also predicted by the physical methods.

10.7. Algorithms

For the purposes of this chapter we will categorize satisfiability algorithms into two broad classes: DPLL algorithms and random walk algorithms. Within the former, we will distinguish between backtracking and non-backtracking algorithms. More concretely, the DPLL procedure for satisfiability is as follows:

DPLL(F)

1. Repeatedly satisfy any pure literals and 1-clauses.
 If the resulting formula F' is empty, exit reporting “satisfiable”.
 If a contradiction (0-clause) is generated, exit.
2. Select a variable $x \in F'$ and a value v for x
3. DPLL($F'_{v=x}$)
4. DPLL($F'_{v=1-x}$)

Clearly, different rules for performing Step 2 give rise to different algorithms and, in practice, the complexity of these rules can vary from minimal to huge. Perhaps the simplest possible rule is to consider the variables in a fixed order, e.g., x_1, x_2, \dots, x_n , always selecting the first variable in the order that is present in the formula, and always setting x to the same value, e.g., $x = 0$. If one forgoes the pure literal rule, something which simplifies the mathematical analysis on random formulas, the resulting algorithms are known as ORDERED DLL. If one also forgoes backtracking, the resulting algorithm is known as UC, for Unit Clause propagation, and was one of the first algorithms to be analyzed on random formulas.

10.7.1. Random walk algorithms

In contrast to DPLL algorithms, random walk algorithms always maintain an entire assignment which they evolve until either it becomes satisfying or the algorithm is exhausted. While, in principle, such an algorithm could switch the value of arbitrarily many variables at a time, typically only a constant number of variables are switched and, in fact, the most common case is to switch only one variable (hence “walk”). An idea that appears to make a significant difference in this context is that of *focusing* [SAO05], i.e., restricting the choice of variable(s) to switch among those contained in clauses violated by the present assignment.

Unfortunately, while there are numerous experimental results regarding performance of random walk algorithms on random formulas, the only mathematically rigorous result known is due to Alekhovich and Ben-Sasson [ABS07] asserting that the algorithm “select a violated clause at random and among its violated

literals select one at random and flip it”, due to Papadimitriou [Pap91], succeeds in linear time on random 3-CNF for densities as high as 1.63, i.e., up to the largest density for which the pure literal rule succeeds (recall that the success of the pure literal implies that the set of satisfying assignments is connected).

It is worth pointing out that, at this point, there exist both very interesting results regarding the performance of random walk algorithms on random formulas [CMMS03, SM03] and also very interesting mathematical results [FMV06, COMV07] regarding their performance on the planted model, i.e., on formulas generated by selecting uniformly at random among all $2^{k-1} \binom{n}{k}$ k -clauses satisfied by a fixed (planted) assignment. Perhaps, our recent understanding of the evolution of the solution-space geometry of random k -CNF formulas will help in transferring some of these results.

10.7.2. Non-backtracking Algorithms

On random CNF formulas UC is equivalent to simply repeating the following until either no clauses remain (success), or a 0-clause is generated (failure): if there is a clause of length 1 satisfy it; otherwise, select a random unassigned variable and assign it a random value. What makes the analysis of UC possible is the fact that if the input is a random k -CNF formula, then throughout the execution of UC the following is true. Let $V(t)$ be the set of variables that have not yet been assigned a value after t steps of the algorithm, and let $\mathcal{C}_i(t)$ be the set of clauses of length i at that time. Then the set $\mathcal{C}_i(t)$ is distributed exactly as a random i -CNF formula on the variables $V(t)$, with exactly $|\mathcal{C}_i(t)|$ clauses.

To see the above claim, imagine representing a CNF formula by a column of k cards for each k -clause, each card bearing the name of one literal. Assume, further, that originally all the cards are “face-down”, i.e., the literal on each card is concealed (and we never had an opportunity to see which literal is on each card). At the same time, assume that an intermediary with photographic memory knows precisely which literal is on each card. To interact with the intermediary we are allowed to either

- Point to a particular card, or,
- Name a variable that has not yet been assigned a value.

In response, if the card we point to carries literal ℓ , the intermediary reveals (flips) all the cards carrying $\ell, \bar{\ell}$. Similarly, if we name variable v , the intermediary reveals all the cards carrying v, \bar{v} . In either case, faced with all the occurrences of the chosen variable we proceed to decide which value to assign to it. Having done so, we remove all the cards corresponding to literals dissatisfied by our setting and all the cards (some of them still concealed) corresponding to satisfied clauses. As a result, at the end of each step only “face-down” cards remain, containing only literals corresponding to unset variables. This card-game representation immediately suggests our claimed “uniform randomness” property for UC and this can be made easily rigorous by appealing to the method of “deferred decisions”. In fact, *all* algorithms that can be carried out via this game enjoy this property.

Lemma 17 (Uniform randomness). *If $V(t) = X$ and $|\mathcal{C}_i(t)| = q_i$, the set of i -clauses remaining at time t form $F_i(|X|, q_i)$ on the variables in X .*

Armed with such a simple representation of Markovian state, it is not hard to show that as long as an algorithm takes care of unit clauses whenever they exist, its success or failure rests on whether the set of 2-clauses ever acquires density greater than 1. The main tool used to determine whether that occurs it is to pass to a so-called “liquid” model and approximate the evolution of the number of clauses of each length via a system of differential equations. Indeed, both of these ideas (uniform randomness plus differential equations) were present in the work of Chao and Franco in the mid-80s [CF86], albeit not fully mathematically justified.

The card-game described above does not allow one to carry out the pure literal heuristic, as it offers no information regarding the number of occurrences of each literal in the formula. To allow for this, we switch to a slightly different model for generating random k -CNF formulas. First, we generate km literals independently, each literal drawn uniformly at random among all $2n$ literals and, then, we generate a formula by partitioning these km literals into k -clauses uniformly at random. (This might result in a few “improper” clauses; we addressed this essentially trivial point in Section 10.6.)

One can visualize the second part of this process as a uniformly random matching between km literals on the left and their km occurrences in k -clauses on the right. As the matching is uniformly random, similarly to the card game, it can be “exposed” on the fly. (Indeed, the card game above is just the result of concealing the left hand of the matching.) With this representation we can now execute algorithms such as: if there is a pure literal or a clause of length 1 satisfy it; otherwise, select a random literal of maximum degree and satisfy it; or, alternatively, select a “most polarized” variable and assign it its majority value. To analyze either one of these algorithms, note that it is enough to maintain as Markovian state the number of clauses of each length and the number of variables having i positive and j negative literal occurrences for each $i, j \geq 0$.

With this long introduction in place we can now describe the state of the art:

- Historically, the first fully rigorous lower bound for r_3 was given by Broder, Frieze and Upfal [BFU93] who considered the pure literal heuristic. They showed that for $r \leq 1.63$, w.h.p. this eventually sets all the variables (and that for $r > 1.7$ w.h.p. it does not). The exact threshold for its success was given later in [LMS98, Mol05].
- Following that, only algorithms expressible in the card-game model were analyzed for a while. In [AS00] Achlioptas and Sorkin showed that the optimal algorithm expressible in this model works up to 3.26... For a survey of the state of the art up to that point, along with a unified framework for analyzing satisfiability algorithms via differential equations, see [Ach01].
- The configuration model for analyzing satisfiability algorithms has been taken up for $k = 3$. Specifically, the simple algorithm mentioned above, “in the absence of unit clauses satisfy a random literal of maximum degree”, was proposed and analyzed by Kaporis, Kirousis and Lalas [KKL02] who showed that it succeeds w.h.p. up to density 3.42... More complicated algorithms that select which literal to satisfy by considering the degree of each literal *and* that of its complement, achieve the best known lower bound, 3.52..., for the 3-SAT threshold, and were analyzed, independently,

by Kaporis, Kirousis and Lalas [KKL06] and Hajiaghayi and Sorkin [HS03]. It seems likely that this bound can be further improved, at least slightly, by making even more refined considerations in the choice of variable and value, but it also seems clear that this is well within the regime of diminishing returns.

- For $k > 3$, the best results come from consider the following very natural “shortest clause” algorithm SC: pick a random clause c of minimum length; among the literals in c pick one at random and satisfy it. A weakening of this algorithm which in the absence of 1-, 2-, and 3-clauses selects a random literal to satisfy was analyzed by Frieze and Suen in [FS96] and gives the best known algorithmic lower bound for the k -SAT threshold for $k > 3$, namely $r_k \geq \gamma_k 2^k/k$, where $\gamma_k \rightarrow 1.817$. In [FS96], the authors give numerical evidence that even the full SC algorithm only succeeds up to some $\zeta_k 2^k/k$, where $\zeta_k = O(1)$.

In a nutshell, the only algorithms that have been proven to find satisfying assignments efficiently in random k -CNF formulas are extremely limited and only succeed for densities up to $O(2^k/k)$. In particular, both their variable ordering and their value assignment heuristic can be implemented given very little, and completely local, information about the variable-clause interactions. Of course, this limitation is also what enables their analysis.

Question 18. *Is there a polynomial-time algorithm which w.u.p.p. finds satisfying assignments of random k -CNF formulas of density $2^k/k^{1-\epsilon}$ for some $\epsilon > 0$?*

As we saw, the geometry of the space of satisfying assignments undergoes a phase transition at $\ln k \cdot 2^k/k$. As a result, an affirmative answer to Question 18 might require a significant departure from the type of algorithms that have been analyzed so far.

10.8. Belief/Survey Propagation and the Algorithmic Barrier

In [MPR02], Mézard, Parisi, and Zecchina proposed a new satisfiability algorithm called Survey Propagation (SP) which performs extremely well experimentally on instances of random 3-SAT. This was a big breakthrough and allowed for optimism that, perhaps, random k -SAT instances might not be so hard, even close to the threshold. Unfortunately, conducting experiments with random k -CNF formulas becomes practically harder at a rapid pace as k increases: the interesting densities scale as $\Theta(2^k)$ so, for example, already $k = 10$ requires extremely large n in order for the formulas to be plausibly considered sparse.

As mentioned earlier, in [KMRT⁺07] it is predicted that just below the satisfiability threshold there is a small range of densities, scaling as $2^k \ln 2 - \Theta(1)$, for which although exponentially many clusters exist, almost all satisfying assignments lie in a *finite* number of (atypically large) clusters. This “condensation” of nearly all satisfying assignments to a small number of clusters induces long-range correlations among the variables, making it difficult to estimate their marginal distributions by examining only a bounded neighborhood around each variable. SP is an ingenious heuristic idea for addressing this problem by considering not

the uniform measure over satisfying assignments but, rather, (an approximation of) the uniform measure over clusters, where each cluster is represented by the fixed point of a certain iterative procedure applied to any assignment in the cluster.

That said, for all densities below the condensation transition, SP is not strictly necessary: if SP can compute variable marginals, then so can a much simpler algorithm called Belief Propagation, i.e., dynamic programming on trees. This is because when the measure is carried by exponentially many well-scattered clusters, marginals are expected to decorrelate. Indeed Gershenfeld and Montanari [GM07] gave very strong rigorous evidence that BP succeeds in computing marginals in the uncondensed regime for the coloring problem. So, although SP might be useful when working very close to the threshold, it is not readily helpful in designing an algorithm that can *provably* find solutions even at *much* lower densities, e.g., say at $r = 2^{k-2}$, roughly in the middle of the satisfiable regime.

One big obstacle is that, currently, to use either BP or SP to find satisfying assignments one sets variables iteratively. When a constant fraction of the variables are frozen in each cluster, as is the case after the dynamical transition, setting a single variable typically eliminates a constant fraction of all clusters. As a result, very quickly, one can be left with so few remaining clusters that decorrelation stops to hold. Concretely, in [MRTS07], Montanari, Ricci-Tersenghi and Semerjian showed that (even with the relatively generous assumptions of statistical physics computations) the following algorithm **fails** for densities greater than $\ln k \cdot 2^k/k$. That is, step 2 below fails to converge after only a small fraction of all variables have been assigned a value:

1. Select a variable v at random.
2. Compute the marginal distribution of v using Belief Propagation.
3. Set v to $\{0, 1\}$ according to the computed marginal distribution; simplify the formula; go to step 1.

10.9. Backtracking Algorithms

Backtracking satisfiability algorithms operate by building an assignment step by step. In particular, the choices they make when no unit clauses and pure literals are present are called *free* and when those choices lead to contradictions (empty clauses), backtracking occurs. In contrast, their non-backtracking variants encountered in Section 10.7.2 simply give up when that happens. Moreover, the only non-backtracking algorithms that have been analyzed maintain their residual formula uniformly random conditional on some small notion of “state”, reflecting the number of clauses of each length and perhaps, additionally, the number of occurrences of each literal. Let us call such algorithms “myopic”.

It is not hard to prove that the largest density, $r_{\mathcal{A}}$, for which a myopic non-backtracking algorithm \mathcal{A} will find satisfying assignments of a random k -CNF formula is precisely the largest density for which its residual 2-CNF subformula remains below density 1 throughout \mathcal{A} 's execution (see e.g. [Ach01]). For densities beyond $r_{\mathcal{A}}$ one can endow \mathcal{A} with a backtracking scheme and attempt to analyze its performance. Unfortunately, any non-trivial amount of backtracking makes it

hard to have a compact probabilistic model for the residual formula. As a result, a probabilistic analysis akin to that possible for $r < r_{\mathcal{A}}$ appears beyond the reach of current mathematical techniques (but see [CM04, CM05, Mon05] for an analysis using techniques of statistical physics). Nevertheless, for *all* backtracking extensions of myopic algorithms it is possible to prove that they take exponential time when the initial k -CNF formula is above a certain critical density. This is because of the following immediate implication of Theorem 9.

Corollary 19. *If a DPLL algorithm ever generates a residual formula that is an unsatisfiable mixture of uniformly random clauses in which the 2-clause density is below 1, then w.h.p. it will spend exponential time before backtracking from it.*

That is, by Corollary 19, once a node in the backtracking search is reached that corresponds to an unsatisfiable random mixture (but where the 2-clauses alone are satisfiable), the search cannot leave the sub-tree for an exponentially long time. Standard results (see e.g. [Ach01]) imply that w.u.p.p. this is precisely what happens for UC started with $3.81n$ 3-clauses and for SC started with $3.98n$ 3-clauses. This is because for such initial densities, at some point, the corresponding algorithm w.u.p.p. generates a residual $(2+p)$ -CNF formula which is unsatisfiable w.h.p. per the results of Theorem 5.

Theorem 20. *For any constant $r \geq 3.81$, any backtracking extension of UC w.u.p.p. takes time $2^{\Omega(n)}$ on $F_3(n, rn)$. Similarly for SC and $r \geq 3.98$.*

We note that the only reason for Theorem 20 is not a high probability result is that w.u.p.p. each algorithm might generate a contradiction and backtrack, thus destroying the uniform randomness of the residual formula, before creating a formula like the one mandated by Corollary 19. It is worth pointing out, though, that whenever this occurs w.h.p. it is for trivial local reasons. In particular, Frieze and Suen in [FS96], introduced the following form of backtracking: when a contradiction is reached, record the portion of the assignment between the last free choice and the contradiction; these literals become *hot*. After flipping the variable involved in the last free choice, instead of making the choice that the original heuristic would suggest, give priority to the complements of the hot literals in the order that they appeared; once the hot literals are exhausted continue as with the original heuristic. This backtracking rule is quite natural in that it is the last part of the partial assignment that got into trouble in the first place. Moreover, it appears to be a genuinely good idea. Experiments on random formulas comparing this backtracking extension of UC with just reversing the last free choice show that the histogram of run-times is *significantly better* for a large range of densities [ABM04b]. Another property of this backtracking rule is that as long as the value of each variable in a partial assignment has been flipped at most once, as happens when dealing with trivial, local contradictions, the residual formula is uniformly random. In particular, for this particular backtracking extension Theorem 20 holds w.h.p.

Theorem 20 sheds light on a widely-cited observation of Mitchell, Selman and Levesque [MSL92], based on experiments with ORDERED-DLL on small problems, stating that random 3-SAT is easy in the satisfiable region up to the 4.2 threshold, becomes sharply much harder at the threshold and quickly becomes easy again at

larger densities in the unsatisfiable region. The upper end of this “easy-hard-easy” characterization is somewhat misleading since, as we saw, the result of [CS88] in fact asserts that w.h.p. random 3-CNF formulas only have exponential-size proofs of unsatisfiability above the threshold. By now, the rate of decline in running time as the density is increased has been analyzed as well by Beame et al. [BKPS02]. Theorem 20 shows that the lower end of this characterization is also misleading in that the onset of exponential behavior occurs significantly below the (conjectured) satisfiability threshold at density 4.2. This concurs with experimental evidence that even the best of current DPLL implementations seem to have bad behavior below the threshold [CDA⁺03]. Moreover, as we will see shortly, the gap between the onset of exponential behavior and the satisfiability threshold increases rapidly with k .

Corollary 19 probably points to a much larger truth than what is specifically derived for the algorithms and backtracking schemes mentioned above. This is because the proof of Theorem 9 is quite robust with respect to the probability distribution of the clauses in the mixture. The essential ingredient seems to be that the variable-clause incidence graph is an expander, suggesting that, in fact, random k -CNF formulas are not the only formulas for which one could hope to prove a result similar to Theorem 20. Moreover, precisely the combinatorial richness of expanders suggests that restarting a DPLL algorithm on a random k -CNF formula is unlikely to yield dramatically different results from run to run, unless, of course, one is willing to restart an exponential number of times.

The bounds on the 3-clause density needed to cause exponential behavior in satisfiability algorithms will be readily improved with any improvement on the $2.28n$ upper bound for unsatisfiability in random $(2+p)$ -SAT. In particular, if Conjecture 6 is true, then Theorem 9 implies [ABM04b] that the running time of every myopic algorithm goes from linear to exponential around a critical, algorithm-specific density.

10.10. Exponential Running-Time for $k > 3$

The most obvious drawback of Theorem 20 is that it implies exponential behavior for densities that are only *conjectured* (but not proven) to be in the “satisfiable regime”. For all $k \geq 4$, the analogue to Theorem 20 holds for densities that are *provably* in the satisfiable regime [ABM04a]. Moreover, the ratio between the density at which the algorithms begins to require exponential time, and the greatest density for which formulas are known to be satisfiable is of order $1/k$. Concretely,

Theorem 21. *For $k = 4$ and $r \geq 7.5$ and for $k \geq 5$ and $r \geq (11/k)2^{k-2}$, ORDERED-DLL w.u.p.p. requires time $2^{\Omega(n)}$ on $F_k(n, rn)$.*

Analogues of Theorem 21 hold for many other backtracking extensions of “myopic” [Ach01] algorithms and, similarly to the results for $k = 3$, one can get high probability results by considering the Frieze-Suen style backtracking.

References

- [ABM04a] D. Achlioptas, P. W. Beame, and M. Molloy. Exponential bounds for dpl below the satisfiability threshold. In *SODA*, pages 139–140, 2004.
- [ABM04b] D. Achlioptas, P. W. Beame, and M. S. O. Molloy. A sharp threshold in proof complexity yields lower bounds for satisfiability search. *J. Comput. Syst. Sci.*, 68(2):238–268, 2004.
- [ABS07] M. Alekhnovich and E. Ben-Sasson. Linear upper bounds for random walk on small density random 3-CNFs. *SIAM J. Comput.*, 36(5):1248–1263, 2007.
- [Ach99] D. Achlioptas. *Threshold phenomena in random graph colouring and satisfiability*. PhD thesis, Toronto, Ont., Canada, Canada, 1999. Adviser-Allan Borodin and Adviser-Michael Molloy.
- [Ach01] D. Achlioptas. Lower bounds for random 3-SAT via differential equations. *Theor. Comput. Sci.*, 265(1-2):159–185, 2001.
- [ACO08] D. Achlioptas and A. Coja-Oghlan. Algorithmic barriers from phase transitions, 2008. preprint.
- [AKKK01] D. Achlioptas, L. M. Kirousis, E. Kranakis, and D. Krizanc. Rigorous results for random $(2 + p)$ -SAT. *Theoret. Comput. Sci.*, 265(1-2):109–129, 2001.
- [Ale05] M. Alekhnovich. Lower bounds for k-dnf resolution on random 3-cnfs. In *STOC '05: Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 251–256, New York, NY, USA, 2005. ACM.
- [AM06] D. Achlioptas and C. Moore. Random k -SAT: Two moments suffice to cross a sharp threshold. *SIAM J. Comput.*, 36(3):740–762, 2006.
- [ANP05] D. Achlioptas, A. Naor, and Y. Peres. Rigorous location of phase transitions in hard optimization problems. *Nature*, 435:759–764, 2005.
- [ANP07] D. Achlioptas, A. Naor, and Y. Peres. On the maximum satisfiability of random formulas. *J. ACM*, 54(2), 2007.
- [AP04] D. Achlioptas and Y. Peres. The threshold for random k -SAT is $2^k \log 2 - O(k)$. *J. Amer. Math. Soc.*, 17(4):947–973, 2004.
- [AR01] M. Alekhnovich and A. A. Razborov. Lower bounds for polynomial calculus: Non-binomial case. In *FOCS*, pages 190–199, 2001.
- [ART06] D. Achlioptas and F. Ricci-Tersenghi. On the solution-space geometry of random constraint satisfaction problems. In *STOC*, pages 130–139. ACM, 2006.
- [AS00] D. Achlioptas and G. B. Sorkin. Optimal myopic algorithms for random 3-sat. In *FOCS*, pages 590–600, 2000.
- [BBC⁺01] B. Bollobás, C. Borgs, J. T. Chayes, J. H. Kim, and D. B. Wilson. The scaling window of the 2-SAT transition. *Random Structures Algorithms*, 18(3):201–256, 2001.
- [BFU93] A. Z. Broder, A. M. Frieze, and E. Upfal. On the satisfiability and maximum satisfiability of random 3-CNF formulas. In *SODA*, pages 322–330, 1993.

- [BKPS02] P. W. Beame, R. Karp, T. Pitassi, and M. Saks. The efficiency of resolution and Davis-Putnam procedures. *SIAM J. Comput.*, 31(4):1048–1075 (electronic), 2002.
- [BMW00] G. Biroli, R. Monasson, and M. Weigt. A variational description of the ground state structure in random satisfiability problems. *Eur. Phys. J. B*, 14:551–568, 2000.
- [BSI99] E. Ben-Sasson and R. Impagliazzo. Random CNF’s are hard for the polynomial calculus. In *FOCS*, pages 415–421, 1999.
- [CDA⁺03] C. Coarfa, D. D. Demopoulos, A. S. M. Aguirre, D. Subramanian, and M. Y. Vardi. Random 3-SAT: The plot thickens. *Constraints*, 8(3):243–261, 2003.
- [CF86] M.-T. Chao and J. Franco. Probabilistic analysis of two heuristics for the 3-satisfiability problem. *SIAM J. Comput.*, 15(4):1106–1118, 1986.
- [CF90] M.-T. Chao and J. Franco. Probabilistic analysis of a generalization of the unit-clause literal selection heuristics for the k -satisfiability problem. *Inform. Sci.*, 51(3):289–314, 1990.
- [CGHS04] D. Coppersmith, D. Gamarnik, M. T. Hajiaghayi, and G. B. Sorkin. Random max sat, random max cut, and their phase transitions. *Random Struct. Algorithms*, 24(4):502–545, 2004.
- [CM97] S. A. Cook and D. G. Mitchell. Finding hard instances of the satisfiability problem: a survey. In *Satisfiability problem: theory and applications (Piscataway, NJ, 1996)*, volume 35 of *DIMACS Ser. Discrete Math. Theoret. Comput. Sci.*, pages 1–17. 1997.
- [CM04] S. Cocco and R. Monasson. Heuristic average-case analysis of the backtrack resolution of random 3-satisfiability instances. *Theoret. Comput. Sci.*, 320(2-3):345–372, 2004.
- [CM05] S. Cocco and R. Monasson. Restarts and exponential acceleration of the Davis-Putnam-Loveland-Logemann algorithm: a large deviation analysis of the generalized unit clause heuristic for random 3-SAT. *Ann. Math. Artif. Intell.*, 43(1-4):153–172, 2005.
- [CMMS03] S. Cocco, R. Monasson, A. Montanari, and G. Semerjian. Approximate analysis of search algorithms with “physical” methods. *CoRR*, cs.CC/0302003, 2003.
- [COGL07] A. Coja-Oghlan, A. Goerdt, and A. Lanka. Strong refutation heuristics for random k -SAT. *Combin. Probab. Comput.*, 16(1):5–28, 2007.
- [COGLS04] A. Coja-Oghlan, A. Goerdt, A. Lanka, and F. Schädlich. Techniques from combinatorial approximation algorithms yield efficient algorithms for random $2k$ -SAT. *Theoret. Comput. Sci.*, 329(1-3):1–45, 2004.
- [COMV07] A. Coja-Oghlan, E. Mossel, and D. Vilenchik. A spectral approach to analyzing belief propagation for 3-coloring. *CoRR*, abs/0712.0171, 2007.
- [CR92] V. Chvátal and B. Reed. Mick gets some (the odds are on his side). In *FOCS*, pages 620–627, 1992.
- [CS88] V. Chvátal and E. Szemerédi. Many hard examples for resolution. *J. Assoc. Comput. Mach.*, 35(4):759–768, 1988.

- [DB97] O. Dubois and Y. Boufkhad. A general upper bound for the satisfiability threshold of random r -SAT formulae. *J. Algorithms*, 24(2):395–420, 1997.
- [DBM03] O. Dubois, Y. Boufkhad, and J. Mandler. Typical random 3-sat formulae and the satisfiability threshold. *Electronic Colloquium on Computational Complexity (ECCC)*, 10(007), 2003.
- [FdIV92] W. Fernandez de la Vega. On random 2-SAT. 1992. Unpublished Manuscript.
- [Fei02] U. Feige. Relations between average case complexity and approximation complexity. In *STOC*, pages 534–543, 2002.
- [FGK05] J. Friedman, A. Goerdt, and M. Krivelevich. Recognizing more unsatisfiable random k -SAT instances efficiently. *SIAM J. Comput.*, 35(2):408–430 (electronic), 2005.
- [FMV06] U. Feige, E. Mossel, and D. Vilenchik. Complete convergence of message passing algorithms for some satisfiability problems. In *APPROX-RANDOM*, pages 339–350, 2006.
- [FP83] J. Franco and M. Paull. Probabilistic analysis of the Davis–Putnam procedure for solving the satisfiability problem. *Discrete Appl. Math.*, 5(1):77–87, 1983.
- [Fra01] J. Franco. Results related to threshold phenomena research in satisfiability: lower bounds. *Theoret. Comput. Sci.*, 265(1-2):147–157, 2001.
- [Fri99] E. Friedgut. Sharp thresholds of graph properties, and the k -SAT problem. *J. Amer. Math. Soc.*, 12:1017–1054, 1999.
- [FS96] A. Frieze and S. Suen. Analysis of two simple heuristics on a random instance of k -SAT. *J. Algorithms*, 20(2):312–355, 1996.
- [GL03] A. Goerdt and A. Lanka. Recognizing more random unsatisfiable 3-SAT instances efficiently. In *Typical case complexity and phase transitions*, volume 16 of *Electron. Notes Discrete Math.*, page 26 pp. (electronic). Elsevier, Amsterdam, 2003.
- [GM07] A. Gerschenfeld and A. Montanari. Reconstruction for models on random graphs. In *FOCS*, pages 194–204, 2007.
- [Goe96] A. Goerdt. A threshold for unsatisfiability. *J. Comput. System Sci.*, 53(3):469–486, 1996.
- [Gol79] A. Goldberg. On the complexity of the satisfiability problem. In *4th Workshop on Automated Deduction (Austin, TX, 1979)*, pages 1–6, 1979.
- [GSCK00] C. P. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *J. Automat. Reason.*, 24(1-2):67–100, 2000.
- [HS03] M. T. Hajiaghayi and G. B. Sorkin. The satisfiability threshold of random 3-SAT is at least 3.52. volume RC22942 of *IBM Research Report*. 2003.
- [KKKS98] L. M. Kirousis, E. Kranakis, D. Krizanc, and Y. Stamatiou. Approximating the unsatisfiability threshold of random formulas. *Random Structures Algorithms*, 12(3):253–269, 1998.
- [KKL02] A. C. Kaporis, L. M. Kirousis, and E. G. Lalas. The probabilistic

- analysis of a greedy satisfiability algorithm. In *Algorithms—ESA 2002*, volume 2461 of *Lecture Notes in Comput. Sci.*, pages 574–585. Springer, Berlin, 2002.
- [KKL06] A. C. Kaporis, L. M. Kirousis, and E. G. Lalas. The probabilistic analysis of a greedy satisfiability algorithm. *Random Structures Algorithms*, 28(4):444–480, 2006.
- [KMRT⁺07] F. Krzakala, A. Montanari, F. Ricci-Tersenghi, G. Semerjian, and L. Zdeborová. Gibbs states and the set of solutions of random constraint satisfaction problems. *Proc. Natl. Acad. Sci. USA*, 104(25):10318–10323 (electronic), 2007.
- [KS94] S. Kirkpatrick and B. Selman. Critical behavior in the satisfiability of random Boolean expressions. *Science*, 264(5163):1297–1301, 1994.
- [LMS98] M. G. Luby, M. Mitzenmacher, and M. A. Shokrollahi. Analysis of random processes via And-Or tree evaluation. In *SODA*, pages 364–373, 1998.
- [MMZ06] S. Mertens, M. Mézard, and R. Zecchina. Threshold values of random K -SAT from the cavity method. *Random Structures Algorithms*, 28(3):340–373, 2006.
- [Mol05] M. S. O. Molloy. Cores in random hypergraphs and Boolean formulas. *Random Structures Algorithms*, 27(1):124–135, 2005.
- [Mon05] R. Monasson. A generating function method for the average-case analysis of DPLL. In *Approximation, randomization and combinatorial optimization*, volume 3624 of *Lecture Notes in Comput. Sci.*, pages 402–413. Springer, Berlin, 2005.
- [MPR02] M. Mézard, G. Parisi, and Z. Ricardo. Analytic and algorithmic solution of random satisfiability problems. *Science*, 297:812 – 815, 2002.
- [MRTS07] A. Montanari, F. Ricci-Tersenghi, and G. Semerjian. Solving constraint satisfaction problems through belief propagation-guided decimation. *CoRR*, abs/0709.1667, 2007.
- [MSL92] D. G. Mitchell, B. Selman, and H. J. Levesque. Hard and easy distributions of sat problems. In *AAAI*, pages 459–465, 1992.
- [MZ98] R. Monasson and R. Zecchina. Tricritical points in random combinatorics: the $(2 + p)$ -SAT case. *J. Phys. A: Math. and Gen.*, 31(46):9209–9217, 1998.
- [MZK⁺] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. Phase transition and search cost in the $(2 + p)$ -SAT problem. In *4th Workshop on Physics and Computation, (Boston, MA, 1996)*.
- [MZK⁺99a] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. $2 + p$ -SAT: relation of typical-case complexity to the nature of the phase transition. *Random Structures Algorithms*, 15(3-4):414–435, 1999. Statistical physics methods in discrete probability, combinatorics, and theoretical computer science (Princeton, NJ, 1997).
- [MZK⁺99b] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. Determining computational complexity from characteristic “phase transitions”. *Nature*, 400(6740):133–137, 1999.
- [Pap91] C. H. Papadimitriou. On selecting a satisfying truth assignment

- (extended abstract). In *FOCS*, pages 163–169. IEEE, 1991.
- [SAO05] S. Seitz, M. Alava, and P. Orponen. Focused local search for random 3-satisfiability. *Journal of Statistical Mechanics: Theory and Experiment*, 6:6–+, June 2005.
- [SM03] G. Semerjian and R. Monasson. A study of pure random walk on random satisfiability problems with "physical" methods. In *SAT*, pages 120–134, 2003.
- [SML96] B. Selman, D. G. Mitchell, and H. J. Levesque. Generating hard satisfiability problems. *Artificial Intelligence*, 81(1-2):17–29, 1996.

Chapter 11

Exploiting Runtime Variation in Complete Solvers

Carla P. Gomes and Ashish Sabharwal

Those working on the satisfiability problem, in particular the designers of highly competitive practical SAT solvers, are faced with an intriguing phenomenon that is both a stumbling block and an opportunity: the performance of backtrack-style complete SAT solvers can vary dramatically depending on “little” details of the heuristics used, such as the way one selects the next variable to branch on (the variable selection heuristic) and in what order the possible values are assigned to the variable (the value selection heuristic). The inherent exponential nature of the search process appears to magnify the unpredictability of such procedures. In fact, it is not uncommon to observe a DPLL-style solver “hang” on a given instance, whereas a different heuristic, or even just another randomized run of the same algorithm, solves the instance within seconds or minutes.

How can one understand this behavior? Even more importantly, how can we devise ways to exploit such extreme variation in the runtime distribution of combinatorial search methods? This chapter focuses on these two questions, in the context of DPLL-style complete search algorithms for SAT.

In the first part of this chapter, we explore runtime variation in a formal manner, characterizing it in terms of certain non-standard classes of statistical distributions. The approach here is to treat combinatorial search methods as complex physical phenomenon, and measure and analyze their performance through statistics obtained over repeated runs. Specifically, this approach reveals that the runtime distribution of many search algorithms often shows what is known as “fat tailed” and “heavy tailed” behavior. Intuitively, this means that the chance that a search procedure will take a certain amount of runtime decays very slowly with the runtime, thereby resulting in an unusually high chance of seeing runs that appear to take forever to terminate. These distributions can have an infinite variance and, in fact, even infinite mean. This is very different from standard distributions like the Gaussian, where the variance is bounded and data points have a significantly low chance of being too far away from the mean. While this provides an explanation for very long runs of SAT solvers that one often observes in practice, a complementary phenomenon—namely, the existence of very small “backdoors” in real-world problem instances—helps us understand how solvers

can sometimes get “lucky” and solve very large complex problems within seconds. The idea here is that if the problem instance under consideration has a small set of variables that capture its key combinatorics, then a well-designed search algorithm has a reasonable chance of discovering this small set and reducing the problem to a much simpler one by assigning truth values to these backdoor variables. Quite surprisingly, instances from many small interesting structured domains have been found to have extremely small backdoor sets (e.g., a few dozen variables in problem instances with thousands of variables).

Randomization-based combinatorial search techniques play a dual role in this setting: revealing and highlighting dramatic runtime variation even on a single instance with a single solver, and opening up ways to address and exploit this variation. Randomization in computational processes, i.e., the ability to decide the next step of an algorithm based on the result of a random experiment, is perhaps one of the most important discoveries in algorithm design. In many problems of interest, randomized algorithms are much simpler than deterministic algorithms of comparable performance, although often the complexity analysis or correctness proofs for randomized algorithms are more involved. The ability to ‘flip a coin’ during computation can make a significant difference in what one can achieve. For each choice of such coin flips, we essentially get a different deterministic algorithm. Thus, executing a randomized algorithm on an input can be viewed as executing a randomly chosen deterministic algorithm on that input. This immediately brings out the strength of randomized algorithms: while it is easy for an adversary to design many worst-case inputs where a given deterministic algorithm fails terribly (in terms of runtime), it is significantly harder to come up with an input where most of the deterministic algorithms represented by a single randomized algorithm fail simultaneously. For a wide selection of randomized algorithms for various combinatorial problems, we refer the reader to Motwani and Raghavan [MR95]. The inherent theoretical power of randomization is the subject of complexity theory, and is discussed in standard texts [e.g. Pap94].

One natural class of techniques for SAT (not covered in this chapter) relying heavily on randomization is that underlying local search solvers, such as *Walksat* [SKC96]. Such algorithms start with an initial candidate solution and make “local” changes to it until either a solution is found or a pre-specified time limit is reached. In order to have any chance of avoiding local “minima” and “traps”, such algorithms rely immensely on randomizing their often greedily biased choice of which variable to flip. Such methods and their history are described in detail in Chapter 6 of this Handbook. We refer the reader to that chapter for an extensive discussion of the use of randomization in local search SAT solvers. Another area involving randomization in SAT is the study of random instances of the k -SAT and other related problem, which has been the subject of extensive research amongst both theoreticians and practitioners, and has revealed intriguing “phase transition” phenomena and “easy-hard-easy” patterns. These aspects are discussed in detail in Chapter 10 of this Handbook.

The present chapter in its second half will largely focus on the use of randomization techniques in DPLL-style backtrack SAT solvers, as a way of addressing the heavy-tailed runtime distributions of such solvers. The presence of heavy-tailed behavior and unusually long runs also implies that there are many runs

that are much shorter than others. The presence of small backdoor sets further indicates that it can—and almost always does—pay off to give up on the current computation of a SAT solver every so often and *restart* the computation from a different state or with a different random seed. This assumes that restarts will send the search in fairly different directions. While there are many opportunities for randomizing a DPLL-style SAT solver which will be mentioned, it turns out the even very simple ones—such as only breaking ties at random—already have a significant desirable effect. In discussing these techniques, we will assume familiarity with standard concepts used in backtrack search for SAT, and refer the reader to Chapter 3 of this Handbook for details.

11.1. Runtime Variation in Backtrack Search

As mentioned earlier, the performance of a randomized backtrack search algorithm can vary dramatically from run to run, even on the same instance. The runtime distributions of backtrack search algorithms are often characterized by what is known as heavy-tailed behavior. This section discusses this behavior and provides formal probabilistic models that help us understand such non-standard distributions.

We note that a related phenomenon is observed in random problem distributions that exhibit an “easy-hard-easy” pattern in computational complexity, concerning so-called “exceptionally hard” instances: such instances seem to defy the “easy-hard-easy” pattern. They occur in the under-constrained area, but they seem to be considerably harder than other similar instances and even harder than instances from the critically constrained area. This phenomenon was first identified by Hogg and Williams [HW94] in the graph coloring problem and by Gent and Walsh [GW94] in the satisfiability problem. An parameterized instance is considered to be exceptionally hard for a particular search algorithm when it occurs in the parameterization region where almost all problem instances are satisfiable (i.e., the under-constrained area), but is considerably harder to solve than other similar instances, and even harder than most of the instances in the critically constrained area [GW94, HW94, SG95]. However, subsequent research showed that such instances are not inherently difficult; for example, by simply renaming the variables or by considering a different search heuristic such instances can be easily solved [SK96, SG97]. Therefore, the “hardness” of exceptionally hard instances does not reside in the instances *per se*, but rather in the combination of the instance with the details of the search method. This is the reason why researchers studying the hardness of computational problems use the median to characterize search difficulty, instead of the mean; the behavior of the mean tends to be quite erratic [GSC97].

11.1.1. Fat and Heavy Tailed Behavior

The study of the full runtime distributions of search methods—instead of just the moments and median—has been shown to provide a better characterization of search methods and much useful information in the design of algorithms. In particular, researchers have shown that the runtime distributions of complete

backtrack search methods reveal intriguing characteristics of such search methods: quite often complete backtrack search methods exhibit *fat* and *heavy-tailed* behavior [FRV97, GSC97, HW94]. Such runtime distributions can be observed when running a deterministic backtracking procedure on a distribution of random instances, and perhaps more importantly, by repeated runs of a randomized backtracking procedure on a *single* instance.

Heavy-tailed distributions were first introduced by the Italian-born Swiss economist Vilfredo Pareto in the context of income distribution. They were extensively studied mathematically by Paul Lévy in the period between the world wars. Lévy worked on a class of probability distributions with heavy tails, which he called *stable* distributions. At the time, however, these distributions were largely considered probabilistic curiosities or pathological cases, mainly used as counterexamples. This situation changed dramatically with Mandelbrot's work on fractals. In particular, two seminal papers by Mandelbrot were instrumental in establishing the use of stable distributions for modeling real-world phenomena [Man60, Man63].

Relatively recently, heavy-tailed distributions have been used to model phenomena in areas as diverse as economics, statistical physics, and geophysics. In particular, they have been applied to stock-market analysis, Brownian motion, weather forecasting, and earthquake prediction — and even for modeling of time delays on the World Wide Web [cf. AFT98, Man83, ST94].

Let us begin with the notion of *fat-tailedness*, which is based on the concept of *kurtosis*. The *kurtosis* is defined as μ_4/μ_2^2 (μ_4 is the fourth central moment about the mean and μ_2 is the second central moment about the mean, i.e., the variance). If a distribution has a high central peak and long tails, then the kurtosis is in general large. The *kurtosis* of the standard normal distribution is 3. A distribution with a *kurtosis* larger than 3 is *fat-tailed* or *leptokurtic*. Examples of distributions that are characterized by *fat-tails* are the exponential distribution, the log-normal distribution, and the Weibull distribution.

Heavy-tailed distributions have “heavier” tails than fat-tailed distributions; in fact they have some infinite moments. For instance, they can have infinite mean, infinite variance, etc. More rigorously, a random variable X with probability distribution function $F(\cdot)$ is heavy-tailed if it has the so-called Pareto like decay of the tails, i.e.:

$$1 - F(x) = \Pr[X > x] \sim Cx^{-\alpha}, \quad x > 0,$$

where $\alpha > 0$ and $C > 0$ are constants. The constant α is called the *index of stability* of the distribution, because which moments of X (if any) are finite is completely determined by the tail behavior. Note that $\alpha = \inf\{r > 0 : \mathbb{E}[X^r] = \infty\}$; hence all the moments of X which are of order strictly less than α are finite, while those of higher order ($\geq \alpha$) are infinite. For example, when $1 < \alpha < 2$, X has infinite variance, and infinite mean and variance when $0 < \alpha \leq 1$.

Like *heavy-tailed* distributions, *fat-tailed* distributions have long tails, with a considerable mass of probability concentrated in the tails. Nevertheless, the tails of *fat-tailed* distributions are lighter than *heavy-tailed* distributions. The log-log plot of $1 - F(x)$ (i.e., the survival function) of a heavy-tailed distribution shows approximately linear behavior with slope determined by α . This is in

contrast to a distribution that decays exponentially, where the tail of a log-log plot should show a faster-than-linear decrease. This property can be used as a visual check for the index of stability of a problem. An example of this is shown in Figure 11.1. The top panel of the figure shows the log-log plot of the survival function, $1 - F(x)$, of the runtime distribution of a quasigroup completion problem (QCP) instance.¹ The log-log plot remains essentially linear till over a hundred thousand backtracks, indicating heavy-tailed behavior. In contrast, the bottom panel of the figure shows a clear faster-than-linear decrease within a few hundred backtracks, indicating that the runtime distribution is not heavy-tailed.

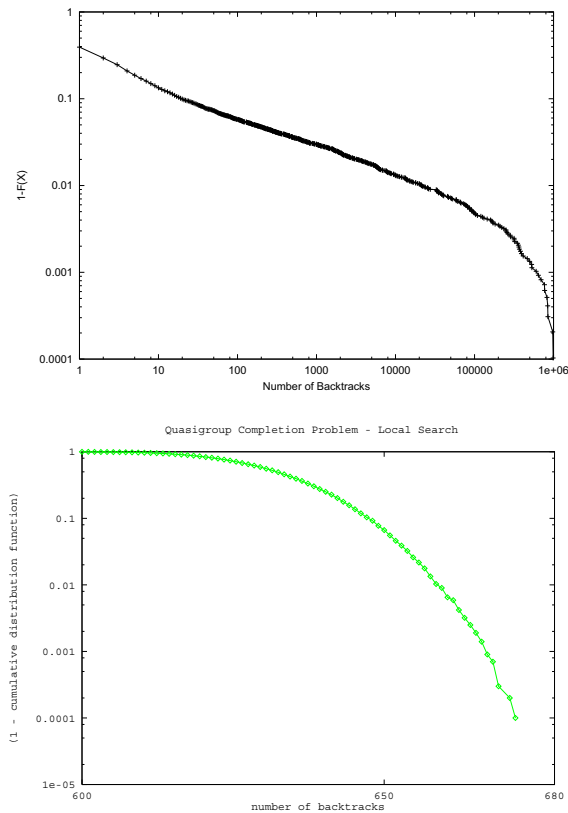


Figure 11.1. Top: log-log plot with heavy-tailed behavior. Bottom: no heavy-tailed behavior.

DPLL style complete backtrack search methods have been shown to exhibit heavy-tailed behavior, both in random instances and real-world instances. Example domains are QCP [GSC97], scheduling [GSMT98], planning [GSK98], model checking, and graph coloring [JM04, Wal99]. If the runtime distribution of a backtrack search method is heavy-tailed, it will produce runs spanning over several

¹The specific details of the problem are not important. For completeness, we mention that this is an instance of order 11 with 30% pre-assigned entries.

orders of magnitude, some extremely long but also some extremely short. Methods like randomization and restarts try to exploit this phenomenon [GSK98]. Several formal models generating heavy-tailed behavior in search have been proposed [CGS01, GFSB04, JM04, WGS03a, WGS03b]. We discuss next one such model based on tree search.

11.1.1.1. Tree Search Model for Heavy Tails

Heavy-tailed behavior in backtrack-style search arises essentially from the fact that wrong branching decisions may lead the procedure to explore an exponentially large subtree of the search space that contains no solutions. Depending on the number of such “bad” branching choices, one can expect considerable variation in the time to find a solution from one run to another. Heavy-tailed behavior has been shown not to be inherent to backtrack search in general, but rather to depend on the structure of the underlying tree search and on the *pruning* power of the heuristics [CGS01].

In the following analysis, we contrast two different tree-search models: a balanced model that does not exhibit heavy-tailed behavior, and an *imbalanced* model that does. A key component of the imbalanced model is that it allows for highly irregular and imbalanced trees, which are radically different from run to run. For the imbalanced tree-search model, one can formally show that the runtime of a randomized backtrack search method is heavy tailed for a range of values of the model parameter p , which characterizes the effectiveness of the branching heuristics and pruning techniques. The heavy-tailedness leads to a runtime distribution with an infinite variance, and sometimes an infinite mean.

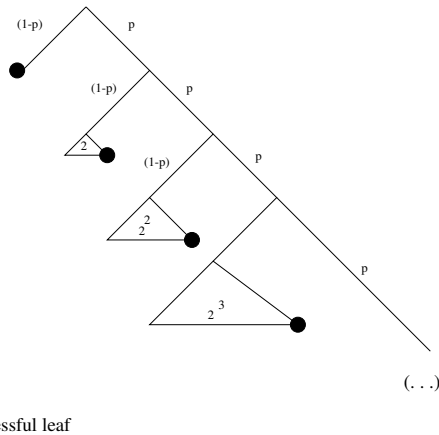


Figure 11.2. The imbalanced tree model with $b = 2$. This tree model has a finite mean and an infinite variance when $1/b^2 < p < 1/b$; both the mean and the variance are infinite when $p \geq 1/b$.

The imbalanced tree model is depicted in Figure 11.2. The model assumes that the probability that the branching heuristic will make a wrong decision is p , hence that, with probability $(1 - p)$ the search is guided directly to a solution.

With probability $p(1-p)$, a search space of size b , with $b \geq 2$, needs to be explored. In general, with probability $p^i(1-p)$, a search space of b^i nodes needs to be explored. Intuitively, p is the probability that the overall amount of backtracking increases geometrically by a factor of b . This increase in backtracking is modeled as a global phenomenon. The larger b and p are, the “heavier” the tail. Indeed, when b and p are sufficiently large, so that $bp \geq 1$, the expected value of T (where T is the number of leaf nodes visited, up to and including the successful leaf node) is infinite: $\mathbb{E}[T] = \infty$. If, however, $bp < 1$ (“better search control”), we obtain a finite mean of $\mathbb{E}[T] = (1-p)/(1-pb)$. Similarly, if $b^2p > 1$ the variance of T is infinite; otherwise, it is finite.

Bounded Heavy-Tailed Behavior for Finite Distributions. The imbalanced tree-search model does not put an *a priori* bound on the size of the search space. In practice, however, the runtime of a backtrack search procedure is bounded above by some exponential function of the size of the instance. We can adjust the model by considering heavy-tailed distributions with bounded support—“bounded heavy-tailed distributions”, for short [cf. HBCM98]. The analysis of the bounded case shows that the main properties of the runtime distribution observed for the unbounded, imbalanced search model have natural analogues when dealing with finite but exponential-sized search spaces. Heavy-tailed distributions have infinitely long tails with power-law decay, while bounded heavy-tailed distributions have exponentially long tails with power-law decay. Also, the concept of an infinite mean in the context of a heavy-tailed distribution translates into a mean that is exponential in the size of the input, when considering bounded heavy-tailed distributions.

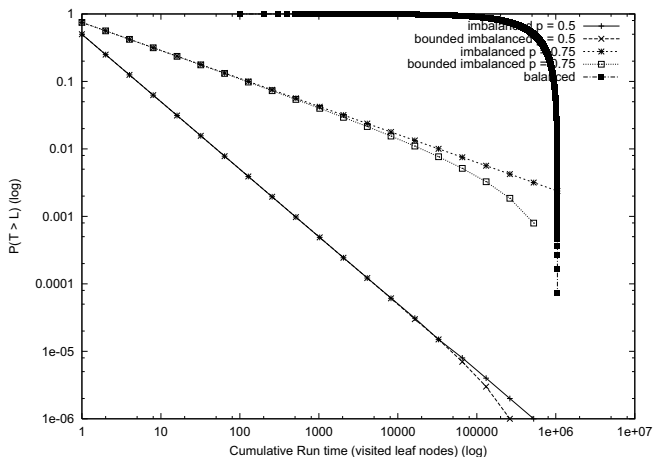


Figure 11.3. Example distributions for the imbalanced and bounded imbalanced models, contrasted with the balanced model. Parameters: $b = 2$, $n = 20$, $p = 0.5$ and 0.75 (n is the number of variables).

Figure 11.3 illustrates and contrasts the distributions for various balanced and imbalanced models. We use the log-log plot of the tails of the various dis-

tributions to highlight the differences between them. The linear behavior over several orders of magnitude for the imbalanced models is characteristic of heavy-tailed behavior. The drop-off at the end of the tail of the distribution for the bounded case illustrates the effect of the boundedness of the search space. However, given the relatively small deviation from the unbounded model (except at the very end of the distribution), we see that the effect of bounding is relatively minor. In the plot we also contrast the behavior of the imbalanced model that of a balanced model (for details, see [CGS01]). The sharp drop-off for the balanced model indicates the absence of heavy-tailedness.

The heavy-tailedness of both of the imbalanced tree-search models occurs as a consequence of the competition between two critical factors: an exponentially increasing penalty in the size of the space to be searched as the number of “mistakes” caused by the branching heuristic increases, and an exponentially decreasing probability that a series of branching mistakes will be made.

11.1.2. Backdoors

Insight into heavy-tailed behavior comes from considering backdoor variables. These are variables which, when set, give us a polynomial subproblem. Intuitively, a small backdoor set explains how a backtrack search method can get “lucky” on certain runs, where backdoor variables are identified early on in the search and set the right way. Formally, the definition of a backdoor depends on a particular algorithm, referred to as *sub-solver*, that solves a tractable sub-case of the general constraint satisfaction problem [WGS03a].

Definition 11.1.1. A *sub-solver* A an algorithm that given as input a Boolean formula F^2 satisfies the following four conditions:

- i. Trichotomy: A either rejects the input F , or “determines” F correctly (as unsatisfiable or satisfiable, returning a solution if satisfiable),
- ii. Efficiency: A runs in polynomial time,
- iii. Trivial solvability: A can determine if F is trivially true (has no clauses) or trivially false (has an empty clause),
- iv. Self-reducibility: if A determines F , then for any variable x and value $v \in \{0, 1\}$, A determines $F[v/x]$.³

For instance, A could be a unit propagation mechanism, a pure literal elimination scheme, a solver for 2-CNF / Horn SAT, an algorithm that enforces arc consistency in CSPs, or, in general, any polynomial time algorithm satisfying the conditions above. Using the definition of sub-solver we can now formally define the concept of a backdoor set for SAT. Let A be a sub-solver and F be a Boolean formula.

²While the notions of sub-solvers and backdoors are applicable to any constraint satisfaction problem, we define them here in the context of SAT.

³We use $F[v/x]$ to denote the simplified formula obtained by fixing the value of variable x to v in F .

Definition 11.1.2. A nonempty subset S of the variables of F is a *weak backdoor* for F w.r.t. A if for some assignment $a_S : S \rightarrow \{0, 1\}$, A returns a satisfying assignment of $F[a_S]$. S is a *strong backdoor* for F w.r.t. A if for all $a_S : S \rightarrow \{0, 1\}$, A returns a satisfying assignment or concludes the unsatisfiability of $F[a_S]$.

Intuitively, the backdoor corresponds to a set of variables, such that when set correctly, the sub-solver can solve the remaining problem. While the notion of weak backdoors is useful for reasoning about satisfiable instances, the notion of strong backdoors considers both satisfiable and unsatisfiable formulas..

Szeider [Sze05] considered the parameterized complexity of the problem of determining whether a SAT instance has a weak or strong backdoor set of size k or less for DPLL style sub-solvers, i.e., subsolvers based on unit propagation and/or pure literal elimination. He showed that detection of weak and strong backdoor sets is unlikely to be fixed-parameter tractable. Nishimura et al. [NRS04] provided more positive results for detecting backdoor sets where the sub-solver solves Horn or 2-CNF formulas, both of which are linear time problems. They proved that the detection of such a strong backdoor set is fixed-parameter tractable, while the detection of a weak backdoor set is not. The explanation that they offered for such a discrepancy is quite interesting: for strong backdoor sets one only has to guarantee that the chosen set of variables gives a subproblem within the chosen syntactic class; for weak backdoor sets, one also has to guarantee satisfiability of the simplified formula, a property that cannot be described syntactically.

Dilkina et al. [DGS07] studied the tradeoff between the complexity of backdoor detection and the backdoor size. They proved that adding certain obvious inconsistency checks to the underlying class can make the complexity of backdoor detection jump from being within NP to being both NP-hard and coNP-hard. On the positive side, they showed that this change can dramatically reduce the size of the resulting backdoors. They also explored the differences between so-called deletion backdoors and strong backdoors, in particular with respect to the class of renamable Horn formulas.

Concerning the size of backdoors, random formulas do not appear to have small backdoor sets. For example, for random 3-SAT problems near the phase transition, the backdoor size appears to be a constant fraction (roughly 30%) of the total number of variables [Int03]. This may explain why the current DPLL based solvers have not made significant progress on hard randomly generated instances. Empirical results based on real-world instances suggest a more positive picture. Structured problem instances can have surprisingly small sets of backdoor variables, which may explain why current state-of-the-art solvers are able to solve very large real-world instances. For example the logistics-d planning problem instance (log.d) has a backdoor set of just 12 variables, compared to a total of nearly 7,000 variables in the formula, using the polynomial time propagation techniques of the SAT solver `Satz` [LA97]. Hoffmann et al. [HGS05] proved the existence of *strong* backdoor sets of size just $O(\log n)$ for certain families of logistics planning problems and blocks world problems.

Even though computing minimum backdoor sets is worst-case intractable [Sze05], if we bound the size of the backdoor, heuristics and techniques like randomization and restarts can often uncover a small backdoor in practice [KSTW05]. In fact, state-of-the-art SAT solvers are surprisingly effective in find-

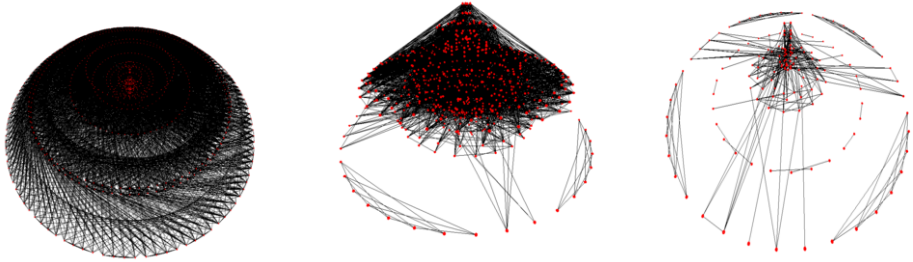


Figure 11.4. Constraint graph of a real-world instance from the logistics planning domain. The instance in the plot has 843 vars and 7,301 clauses. One backdoor set for this instance w.r.t. unit propagation has size 16 (not necessarily the minimum backdoor set). Left: Constraint graph of the original instance. Center: Constraint graph after setting 5 variables and performing unit propagation. Right: Constraint graph after setting 14 variables and performing unit propagation.

ing small backdoors in many structured problem instances. Figure 11.4 shows a visualization of the constraint graph of a logistics planning problem and how this graph is drastically simplified after only a few variables occurring in a small backdoor (found by SAT solvers) are set. In related work, Slaney and Walsh [SW01] studied the structural notion of “backbones” and Dequen and Dubois introduced a heuristic for DPLL based solvers that exploits the notion of backbone and outperforms other heuristics on random 3-SAT problems [DD03, DD01].

11.2. Exploiting Runtime Variation: Randomization and Restarts

As it turns out, one of the most effective ways to address and exploit heavy-tailed behavior is to add “restarts” to a backtracking procedure. We describe this technique next, followed by a brief discussion of various ways to randomize a backtrack search algorithm.

11.2.1. Restarts: Randomized and Deterministic Strategies

A *restart*, as the name suggests, is the process of stopping the current computation of a SAT solver and restarting it from the beginning with a different random seed.⁴ For solvers employ caching techniques such as clause learning, the information learned in one run is kept and used even after a restart, thus not letting the computational effort spent so far go completely waste.

In the presence of heavy tailed behavior, a sequence of short runs instead of a single long run may be a more effective use of computational resources. Gomes et al. [GSK98] proposed randomized rapid restarts (RRR) to take advantage of

⁴As we will see later in Section 11.2.2, restarting can be effective even for a deterministic algorithm when other features such as learned clauses naturally guide the deterministic search process in a different direction after the restart.

heavy-tailed behavior and boost the efficiency of complete backtrack search procedures. In practice, one gradually increases the cutoff to maintain completeness [BMS00, GSK98]. It has been demonstrated that a restart strategy with a fixed cutoff eliminates heavy-tail behavior and has finite moments [GSCK00].

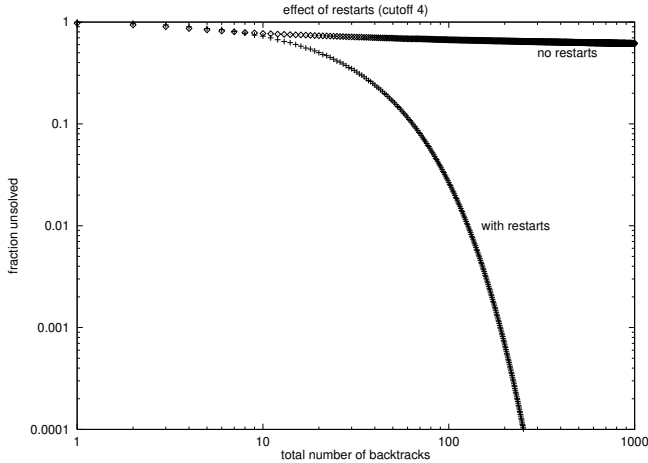


Figure 11.5. Restarts: Tail ($1 - F(x)$) as a function of the total number of backtracks for a QCP instance, log-log scale; the lower curve is for a cutoff value of 4 and the upper curve is without restarts.

As an example, Figure 11.5 shows the result of applying a strategy of fixed-length short runs—*rapid randomized restarts* of a complete randomized search procedure—to a QCP instance of order 20 with 5% pre-assignment. The figure displays a log-log plot of the tail of the distribution. From the figure, we see that without restarts and given a total of 50 backtracks, we have a failure rate of about 70%. Using restarts (once after every 4 backtracks), this failure rate drops to about 10%. With an overall limit of only 150 backtracks, the restart strategy nearly always solves the instance, whereas the original procedure still has a failure rate of about 70%. Such a dramatic improvement due to restarts is typical for heavy-tailed distributions; in particular, we get similar results on critically constrained instances. The fact that the curve for the experiment with restarts takes a definite downward turn is a clear indication that the heavy-tailed nature of the original cost distribution has disappeared.

Figure 11.6 shows the effect on the number of backtracks needed to solve a logistics planning problem as a function of the fixed cutoff value used. From the plot we see that the optimal cutoff value here is around 12. This is, in fact, quite typical of real-world problems, for which the optimal cutoff value is surprisingly small. Further, the logarithmic vertical scale indicates that one can shift the performance of the procedure by several orders of magnitude by tuning the cutoff parameter.

Prior to the discovery of heavy-tailed behavior and backdoor sets, randomized restart policies have been studied in the context of general randomized Las Vegas procedures. Luby et al. [LSZ93] showed that when the underlying runtime

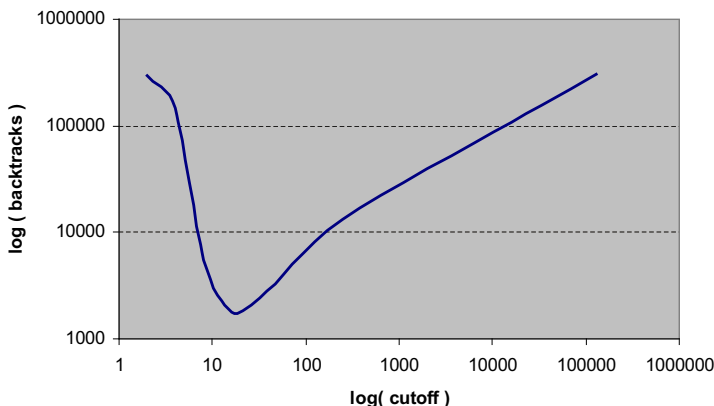


Figure 11.6. Restarts: The effect of different cutoff values on solution cost for the `logistics.d` planning problem.

distribution of the randomized procedure is fully known, the optimal restart policy is a fixed cutoff. When there is no *a priori* knowledge about the distribution, they also provided a *universal strategy* which minimizes the expected cost. This consists of runs whose lengths are powers of two, and each time a pair of runs of a given length has been completed, a run of twice that length is immediately executed. The universal strategy is of the form: 1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 4, 8, . . . Although the universal strategy of Luby et al. is provably within a log factor of the optimal fixed cutoff, the schedule often converges too slowly in practice. Walsh [Wal99] introduced a restart strategy, inspired by Luby et al.'s analysis, in which the cutoff value increases geometrically. The advantage of such a strategy is that it is less sensitive to the details of the underlying distribution. Following the findings of Gomes et al. [GSK98] and starting with `zChaff`, state-of-the-art SAT solvers now routinely use restarts. In practice, the solvers use a default cutoff value, which is increased, linearly, every given number of restarts, guaranteeing the completeness of the solver in the limit [MMZ⁺01]. Another important feature is that they retain learned clauses across restarts.

In reality, we will be somewhere between full and no knowledge of the runtime distribution. Horvitz et al. [HRG⁺01] introduced a Bayesian framework for learning predictive models of randomized backtrack solvers based on this situation. Extending that work, Kautz et al. [KHR⁺02] considered restart policies that can factor in information based on real-time observations about a solver's behavior. In particular, they introduced an *optimal* policy for dynamic restarts that considers observations about solver behavior. They also considered the dependency between runs. They gave a dynamic programming approach to generate the optimal restart strategy, and combined the resulting policy with real-time observations to boost performance of backtrack search methods.

Variants of restart strategies include randomized backtracking [LBMS01] and the random jump strategy [Zha02] which has been used to solve a dozen previously open problems in finite algebra. Finally, one can also take advantage of the high variance of combinatorial search methods by combining several algorithms

into a “portfolio,” and running them in parallel or interleaving them on a single processor [GS01, NDS⁺04].

As a closing note, we mention that the notion of backdoor sets discussed earlier came about in the context of the study of heavy-tailed behavior as observed in backtrack-style search. Heavy-tailed distributions provide a justification of why restarts are effective, namely to prevent the search procedure from getting stuck in unproductive portions of the search space that do not contain solutions. Such distributions also imply the existence of a wide range of solution times, often including short runs. This is where backdoors enter the picture: Intuitively, a small backdoor explains how a backtrack search can get “lucky” on certain runs, since the backdoor variables are identified early on in the search and set in the right way.

Williams et al. show that even though finding a small set of backdoor variables is computationally hard, the very existence of a small backdoor in a problem provides a concrete computational advantage to its solution [WGS03a, WGS03b]. They consider three scenarios. First, a deterministic scenario is considered with an exhaustive search of backdoor sets. In this scenario, one obtains provably better search complexity when the backdoor contains up to a certain fraction of all variables. They then show that a randomized search technique, which in effect repeatedly guesses backdoor sets, provably outperforms a deterministic search. Finally, in the third scenario the availability of a variable selection heuristic is considered, which provides guidance in looking for a backdoor set. This strategy can reduce the search space even further. By exploiting restart strategies, for example, one can obtain a polynomially solvable case when the backdoor contains at most $\log(n)$ variables, where n is the total number of variables in the problem instance. This final scenario is quite likely the closest to the behavior of the effective randomized backtrack SAT solvers that are currently available.

11.2.2. Other Randomization Opportunities in Backtrack Search

There are also several other opportunities to introduce randomization into a backtrack search method, at the different points where the next operation to be performed by the backtrack solver is to be selected. In particular, randomization can be incorporated into the variable selection heuristic, the value selection heuristic, look-ahead procedures such as propagation and failed literal tests, and look-back procedures such as the choice of conflict clause to learn. Even a simple modification to a deterministic SAT solver, such as adding randomized tie-breaking in variable and value selection heuristics, can dramatically change its behavior [GSC97, GSCK00].

In general, backtrack search procedures use *deterministic* heuristics to select the next operation. Highly tuned heuristics are used for variable and value selection. If several choices are heuristically determined to be equally good, then a deterministic algorithm applies some fixed rule to pick one of the operations, for example, using lexicographical order. Therefore, the most straightforward way to apply randomization is in this tie-breaking step: if several choices are ranked equally, choose among them at random [GSC97, GSCK00]. Even this simple modification can dramatically change the behavior of a search algorithm,

as we will see below. If the heuristic function is particularly powerful, however, it may rarely assign the highest score to more than one choice. To handle this, we can introduce a “heuristic-equivalence” parameter to the algorithm. Setting the parameter to a value H greater than zero means all choices that receive scores within H percent of the highest score are considered equally good. This expands the choice set for random tie-breaking [GSK98].

A more general procedure, which includes the tie-breaking and the H -percent equivalence-window procedures as special cases, imposes a probability distribution on the set of possible values for a given operation. For example, in the tie-breaking case, we are implicitly assigning a uniform probability to the values that are considered by the heuristic to be equally good, while in the case of using the H -percent window, we assign a uniform probability to all choices whose scores lie within H percent of the highest score. An alternative would be to impose a different distribution on the values within the H -percent window (for example, an exponential distribution), biasing the selection toward the highest score.

Random reordering is another way of “randomizing” an algorithm. This technique involves randomly reordering the input data, followed by the application of a deterministic algorithm [MR95]. Walsh [Wal99] applied this technique to study the runtime distributions of graph-coloring problems, using a deterministic, exact coloring algorithm based on DSATUR [Tri96].

A key issue when introducing randomization into a backtrack search algorithm is that of guaranteeing completeness. We note that the introduction of randomness into the branching-variable/value selection does not affect the completeness of the backtrack search. Some basic bookkeeping ensures that the procedures do not revisit any previously explored part of the search space, which means that, unlike the use of local search methods, we can still determine inconsistencies. The bookkeeping mechanism involves keeping track of some additional information for each variable in the stack, namely which assignments have been tried thus far. This can be done relatively inexpensively.

Randomization of the look-ahead and look-back procedures can also be done in a relatively straightforward manner, for example, by randomizing the decision of whether to apply such procedures after each variable/value assignment. This technique can be useful, especially when look-ahead techniques are expensive. On the other hand, randomization of the backtrack points requires the use of data structures that are more complicated in general, so there may be a substantial increase in the time/space requirements of the algorithm. Lynce *et al.* [LBMS01, LMS02] discuss learning strategies and randomization of the backtrack points (specifically, random backtracking and unrestricted backtracking), as well as how to maintain completeness).

We should note that when implementing a randomized algorithm, one uses a pseudo random-number generator, which is in fact a deterministic algorithm for generating “random” numbers. By choosing different initial random seeds, we obtain different runs of the algorithm. For experimental purposes, it is important to save the “seed” given to the random-number generator, so that the same experiment can be replayed. Also, when using restarts, it is important to pass on the seed from one run to another (rather than generating a new seed for each restart) in order to use the full strength of the pseudo random-number generator.

One can also speak of “deterministic randomization” [Wol02], which expresses the fact that the behavior of some very complex deterministic systems is so unpredictable that it actually appears to be random. This notion of deterministic randomization is implicitly used, for example, in calculating “random” backtrack points by applying a deterministic formula to the clauses learned during search [LBMS01, LMS02]. Another example is the restarting of (deterministic) backtrack search solvers that feature clause learning: each time the solver is restarted, it behaves quite differently from the previous run (because of the additional learned clauses), and thus appears to behave “randomly” [MMZ⁺01, LMS02].

11.3. Conclusion

The heavy-tailed nature of the runtime distribution of SAT solvers, and of constraint solvers in general, on many problem instances of interest has been an insightful discovery with a practical impact. The theory behind such distributions, as well as concepts such as backdoor sets, help us understand the brittle nature of combinatorial search algorithms, where even the most efficient implementations seem to “hang” forever with certain heuristic settings and solve the same instance quickly with other settings or with simple random renaming of the instance variables. This has led to an interest in randomizing SAT solvers and in introducing very frequent restart schedules, both of which have generally been quite successful. Aggressive restart strategies are now an integral part of nearly all DPLL-style SAT solvers and some of the latest solvers, such as `Rsat` [PD06], are beginning to explore interesting new ways of retaining information across restarts.

References

- [AFT98] R. J. Adler, R. E. Feldman, and M. Taqqu. *A practical guide to heavy tails*. Birkhäuser, 1998.
- [BMS00] L. Baptista and J. P. Marques-Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In *CP-00: 6th International Conference on Principles and Practice of Constraint Programming*, pages 489–494, Singapore, September 2000.
- [CGS01] H. Chen, C. P. Gomes, and B. Selman. Formal models of heavy-tailed behavior in combinatorial search. In *CP-01: 7th International Conference on Principles and Practice of Constraint Programming*, 2001.
- [DD01] O. Dubois and G. Dequen. A backbone-search heuristic for efficient solving of hard 3-SAT formulae. In *Proceedings of IJCAI-01: 17th International Joint Conference on Artificial Intelligence*, 2001.
- [DD03] G. Dequen and O. Dubois. Kcnfs: An efficient solver for random k-SAT formulae. In *Proceedings of SAT-03: 6th International Conference on Theory and Applications of Satisfiability Testing*, 2003.
- [DGS07] B. Dilkina, C. P. Gomes, and A. Sabharwal. Tradeoffs in the complexity of backdoor detection. In *CP-07: 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741

- of *Lecture Notes in Computer Science*, pages 256–270, Providence, RI, September 2007.
- [FRV97] D. Frost, I. Rish, and L. Vila. Summarizing CSP hardness with continuous probability distributions. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 327–334, New Providence, RI, 1997. AAAI Press.
- [GFSB04] C. P. Gomes, C. Fernández, B. Selman, and C. Bessière. Statistical regimes across constrainedness regions. In *CP-04: 10th International Conference on Principles and Practice of Constraint Programming*, 2004.
- [GS01] C. P. Gomes and B. Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1-2):43–62, 2001.
- [GSC97] C. P. Gomes, B. Selman, and N. Crato. Heavy-tailed distributions in combinatorial search. In *CP-97: 3rd International Conference on Principles and Practice of Constraint Programming*, pages 121–135, 1997.
- [GSCK00] C. P. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24(1/2):67–100, 2000.
- [GSK98] C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings of AAAI-98: 15th National Conference on Artificial Intelligence*, pages 431–437, Madison, WI, July 1998.
- [GSMT98] C. P. Gomes, B. Selman, K. McAloon, and C. Tretkoff. Randomization in backtrack search: Exploiting heavy-tailed profiles for solving hard scheduling problems. In *Proceedings of AIPS-98: 4th International Conference on Artificial Intelligence Planning Systems*, 1998.
- [GW94] I. P. Gent and T. Walsh. Easy problems are sometimes hard. *Artificial Intelligence*, 70:335–345, 1994.
- [HBCM98] M. Harchol-Balter, M. E. Crovella, and C. D. Murta. On choosing a task assignment policy for a distributed server system. In *Proceedings of Performance Tools '98*, pages 231–242. Springer-Verlag, 1998.
- [HGS05] J. Hoffmann, C. P. Gomes, and B. Selman. Structure and problem hardness: Asymmetry and DPLL proofs in SAT-based planning. In *CP-05: 11th International Conference on Principles and Practice of Constraint Programming*, 2005.
- [HRG⁺01] E. Horvitz, Y. Ruan, C. P. Gomes, H. Kautz, B. Selman, and D. Chickering. A bayesian approach to tackling hard computational problems. In *Proceedings of UAI-01: 17th Conference on Uncertainty in Artificial Intelligence*, 2001.
- [HW94] T. Hogg and C. Williams. Expected gains from parallelizing constraint solving for hard problems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pages 1310–1315, Seattle, WA, 1994. AAAI Press.
- [Int03] Y. Interian. Backdoor sets for random 3-SAT. In *Proceedings of SAT-03: 6th International Conference on Theory and Applications of Satisfiability Testing*, 2003.

- [JM04] H. Jia and C. Moore. How much backtracking does it take to color random graphs? rigorous results on heavy tails. In *CP-04: 10th International Conference on Principles and Practice of Constraint Programming*, 2004.
- [KHR⁺02] H. Kautz, E. Horvitz, Y. Ruan, C. P. Gomes, and B. Selman. Dynamic restart policies. In *Proceedings of AAAI-02: 18th Conference on Artificial Intelligence*, 2002.
- [KSTW05] P. Kilby, J. Slaney, S. Thiébaux, and T. Walsh. Backbones and backdoors in satisfiability. In *Proceedings of AAAI-05: 20th National Conference on Artificial Intelligence*, 2005.
- [LA97] C. M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of IJCAI-97: 15th International Joint Conference on Artificial Intelligence*, 1997.
- [LBMS01] I. Lynce, L. Baptista, and J. Marques-Silva. Stochastic systematic search algorithms for satisfiability. In *Proceedings of SAT-01: 4th International Conference on Theory and Applications of Satisfiability Testing*, 2001.
- [LMS02] I. Lynce and J. Marques-Silva. Complete unrestricted backtracking algorithms for satisfiability. In *Fifth International Symposium on the Theory and Applications of Satisfiability Testing (SAT'02)*, 2002.
- [LSZ93] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47:173–180, 1993.
- [Man60] B. Mandelbrot. The Pareto-Lévy law and the distribution of income. *International Economic Review*, 1:79–106, 1960.
- [Man63] B. Mandelbrot. The variation of certain speculative prices. *Journal of Business*, 36:394–419, 1963.
- [Man83] B. Mandelbrot. *The fractal geometry of nature*. Freeman: New York, 1983.
- [MMZ⁺01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of DAC-01: 38th Design Automation Conference*, pages 530–535, Las Vegas, NV, June 2001.
- [MR95] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
- [NDS⁺04] E. Nudelman, A. Devkar, Y. Shoham, K. Leyton-Brown, and H. H. Hoos. SATzilla: An algorithm portfolio for SAT, 2004. In conjunction with SAT-04.
- [NRS04] N. Nishimura, P. Ragde, and S. Szeider. Detecting backdoor sets with respect to Horn and binary clauses. In *Proceedings of SAT-04: 7th International Conference on Theory and Applications of Satisfiability Testing*, 2004.
- [Pap94] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
- [PD06] K. Pipatsrisawat and A. Darwiche. RSat 1.03: SAT solver description. Technical Report D-152, Automated Reasoning Group, Computer Science Department, UCLA, 2006.
- [SG95] B. M. Smith and S. A. Grant. Sparse constraint graphs and exception-

- ally hard problems. In *Proceedings of IJCAI-95: 14th International Joint Conference on Artificial Intelligence*, volume 1, pages 646–654, Montreal, Canada, August 1995.
- [SG97] B. M. Smith and S. A. Grant. Modelling exceptionally hard constraint satisfaction problems. In *CP-97: 3rd International Conference on Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 182–195, Austria, October 1997.
- [SK96] B. Selman and S. Kirkpatrick. Finite-Size Scaling of the Computational Cost of Systematic Search. *Artificial Intelligence*, 81(1–2):273–295, 1996.
- [SKC96] B. Selman, H. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In D. S. Johnson and M. A. Trick, editors, *Cliques, Coloring and Satisfiability: the Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 521–532. American Mathematical Society, 1996.
- [ST94] G. Samorodnitsky and M. Taqqu. *Stable Non-Gaussian Random Processes: Stochastic Models with Infinite Variance*. Chapman and Hall, 1994.
- [SW01] J. K. Slaney and T. Walsh. Backbones in optimization and approximation. In *Proceedings of IJCAI-01: 17th International Joint Conference on Artificial Intelligence*, pages 254–259, Seattle, WA, August 2001.
- [Sze05] S. Szeider. Backdoor sets for DLL subsolvers. *Journal of Automated Reasoning*, 35(1-3):73–88, 2005. Special issue on SAT 2005.
- [Tri96] M. Trick. <http://mat.gsia.cmu.edu/COLOR/Solvers/trick.c>. Source code for the implementation of an exact deterministic algorithm for graph coloring based on DSATUR, 1996.
- [Wal99] T. Walsh. Search in a small world. In *Proceedings of IJCAI-99: 16th International Joint Conference on Artificial Intelligence*, 1999.
- [WGS03a] R. Williams, C. P. Gomes, and B. Selman. Backdoors to typical case complexity. In *Proceedings of IJCAI-03: 18th International Joint Conference on Artificial Intelligence*, 2003.
- [WGS03b] R. Williams, C. P. Gomes, and B. Selman. On the connections between backdoors, restarts, and heavy-tailedness in combinatorial search. In *Proceedings of SAT-03: 6th International Conference on Theory and Applications of Satisfiability Testing*, 2003.
- [Wol02] S. Wolfram. *A New Kind of Science*. Stephen Wolfram, 2002.
- [Zha02] H. Zhang. A random jump strategy for combinatorial search. In *International Symposium on AI and Math*, Fort Lauderdale, FL, 2002.

Chapter 12

Automated Configuration and Selection of SAT Solvers

Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown

12.1. Introduction

Although SAT is widely believed to be hard in the worst case, many interesting types of SAT instances can be solved efficiently in practice, using heuristic algorithms. A wide variety of different heuristic mechanisms have been identified by the research community over the decades, partly by unambiguously improving on past ideas, but largely by finding different heuristics that work well on different types of SAT instances. Because of this sensitivity to instance characteristics, there is no single best SAT solver, and state-of-the-art solvers typically expose some design choices as parameters that can be set at runtime, thereby enabling users to optimize performance to a given class of SAT instances. This chapter focuses on automated approaches for performing such optimization, which is termed *algorithm configuration*. We also discuss the related problem of *algorithm selection*: i.e., methods for choosing one algorithm from a finite, typically modestly sized *portfolio* on a per-instance basis, with the goal of achieving better performance across instances than can be obtained from any single solver in the portfolio.

Automated algorithm configuration and per-instance algorithm selection are both examples of *meta-algorithmic design techniques*, i.e., generic methods for building algorithms for a given problem from a set or space of candidate algorithms (which can be small, as in the case of algorithm selection, or large, as in the case of algorithm configuration). Automated algorithm configuration and per-instance algorithm selection are of particular interest in the context of SAT, because of their major impact on solving SAT and related problems, such as MaxSAT, ASP, CSP and QBF. Conversely, SAT has played a major role in the development of algorithm configuration and selection techniques.

While the major part of this chapter is focused on automated algorithm configuration and per-instance algorithm selection, we also survey some conceptually more complex approaches that go beyond “pure” configuration and selection: combining configuration and selection together; building algorithm schedules rather than simply picking a single algorithm; extending methods to leverage

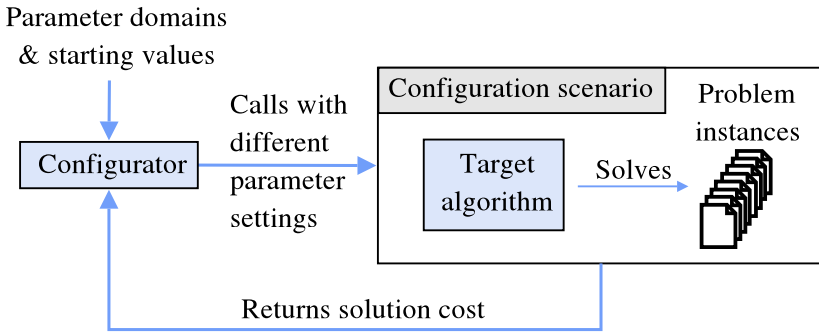


Figure 12.1. A configuration scenario includes an algorithm to be configured and a collection of problem instances. A configuration procedure executes the target algorithm with specified parameter settings on some or all of the instances, receives information about the performance of these runs, and uses this information to decide which subsequent parameter configurations to evaluate.

parallel computation; analyzing the importance of parameters; dynamically controlling parameters; and using configuration techniques for finding bugs in SAT solvers. Finally, we outline some open challenges related to the methods discussed in this chapter, the tools based on them, and their application.

12.2. Algorithm configuration

Algorithm configuration is the problem of automatically identifying parameter settings that optimize the performance of a given algorithm on a specific set or distribution of problem instances, as illustrated in Figure 12.1. Observe that an algorithm configuration procedure is a *meta-algorithm* in the sense that it takes another, arbitrary algorithm as part of its input.

More formally, the algorithm configuration problem can be stated as follows: given an algorithm A with possible parameter settings $\Theta = \Theta_1 \times \dots \times \Theta_n$, a distribution \mathcal{D} over possible inputs with domain Π that A can be run on, and a cost metric $c : \Theta \times \Pi \rightarrow \mathbb{R}$, find the configuration $\theta = \langle \theta_1, \dots, \theta_n \rangle \in \Theta$ that minimizes c in expectation over \mathcal{D} . A common special case arises where \mathcal{D} is a uniform distribution over $\Pi = \{\pi_1, \dots, \pi_k\}$, in which case we can write the problem as finding the minimizer of the blackbox function $f(\theta) := \frac{1}{k} \sum_{i=1}^k c(\theta, \pi_i)$. When applying algorithm configuration to SAT solvers, the π_i are SAT instances deemed representative for the kind of instances for which we aim to optimize performance. We call f a blackbox function, because it depends on the cost $c(\theta, \pi_i)$, which cannot be reasoned about analytically, but can only be evaluated by actually running A with configuration θ on π_i .

Although the problem of “tuning” algorithm parameters might seem narrow, we note that many aspects of algorithm design can be expressed parametrically, from high-level design decisions to low-level aspects of specific heuristic mecha-

nisms. For example, parameters can be real-valued, integer-valued, or categorical. Parameters can also be conditional on an instantiation of other parameters—consider, e.g., the parameters of a heuristic mechanism h , which are completely ignored unless h is enabled via another, categorical parameter. Finally, combinations of otherwise valid parameter instantiations can be forbidden.

Indeed, the flexibility offered by expressing heuristic ideas parametrically and the effectiveness of algorithm configuration procedures can empower algorithm developers to adopt a new design philosophy that is both easier and produces better results. Instead of seeking to minimize the number of exposed parameters, a developer can instead aim to encode all plausible design ideas parametrically, leaving their eventual evaluation to the algorithm configuration procedure (see, e.g., [HBHH07, KXHLB09, KXHLB16]). This philosophy has been formalized in a design paradigm called Programming by Optimization (PbO), which facilitates the exploration and automatic optimization of design choices as an integral part of the software development process [Hoo12].

Over the past decade, evidence has accumulated that algorithm configuration is an effective way to design high-performance algorithms across a wide range of applications. This is because human experts are not only ill-suited for solving these noisy high-dimensional optimization problems manually, but often also unmotivated to carry out this tedious task (not to mention that experts rarely have time and are also expensive in practice). Our focus in this chapter is on SAT; nevertheless, it is important to point out that automated algorithm configuration has been demonstrated to produce dramatic improvements in a wide range of other combinatorial optimization problems beyond SAT, such as MIP [HHLB10], AI planning [VFG⁺13], TSP [CFH08], and answer set programming [SLS12]; these improvements routinely exceed those achieved manually by human experts. Likewise, very similar automated approaches have yielded better performance than humans for optimizing hyperparameters of deep convolutional neural networks [BBBK11, SLA12] and their architectures [DSH15], as well as generic image recognition pipelines [BYC14]; they have even led to effective push-button approaches for machine learning, by jointly selecting feature selection methods, algorithms and hyperparameter settings in prominent machine learning packages [THHLB13, FKE⁺15].

Automated algorithm configuration approaches also have benefits beyond optimizing algorithm performance. They can additionally be used to speed up the process of developing effective software (freeing developers from the burden of tedious experimentation); to apply highly parameterized algorithms to a new problem domain; and to improve protocols for empirical research in computer science (enabling fair comparisons of software based on the same fully formalized parameter optimization approach).

In the remainder of this section, we briefly survey some widely used algorithm configuration procedures—particularly those that have been widely applied to SAT. Throughout, we emphasize key conceptual ideas and methods.

12.2.1. ParamILS

The ParamILS framework [HHS07, HHLBS09] was the first to give rise to practical, general-purpose algorithm configuration procedures; it still yields excel-

lent performance on many problems today. This approach uses iterated local search [LMS02] as its underlying search strategy. It begins with a default parameter configuration, evaluates performance against a list of random configurations, and then performs local search in configuration space, starting from the best configuration so far. The local search process changes the setting of one parameter at a time and keeps those changes resulting in empirical performance improvements. After finding a local minimum, ParamILS randomly changes some parameters in order to escape; with a small probability, it also performs a complete restart to a random point in the configuration space. Throughout, ParamILS keeps track of the best configuration seen so far, the so-called *incumbent*.

ParamILS has two variants that differ in the approach they take to deciding how many runs to use for estimating the performance of each configuration. The *BasicILS* variant is most straightforward: it evaluates each configuration according to a fixed number of runs on a fixed set of instances. However, this variant has the drawback that it can spend far too long evaluating configurations that exhibit poor overall performance. The more practical *FocusedILS* variant starts by performing only a single run per configuration; if a new configuration does not outperform the incumbent on the same instance, it is rejected. If it beats the incumbent, FocusedILS doubles the number of instances to consider and tries again, until eventually it evaluates the new configuration on the same set of instances as the incumbent has been evaluated on. The number of runs allocated to incumbents also grows slowly over time. Overall, FocusedILS thus devotes most of its time to promising configurations, while also maintaining statistically robust performance estimates for incumbent configurations.

Both ParamILS variants can be sped up via an *adaptive capping* strategy for choosing the time after which to terminate unsuccessful target algorithm runs. Intuitively, when comparing the performance of two configurations θ_1 and θ_2 on an instance, and we already know that θ_1 solves the instance in time t_1 , we do not need to run θ_2 for longer than t_1 : we do not need to know precisely how bad θ_2 is, as long as we know that θ_1 is better. While this strategy exactly preserves the search trajectory of ParamILS and leads to substantial speedups, it does not perform effectively after a perturbation step. A more aggressive adaptive capping variant terminates all evaluations prematurely if they require more time than the corresponding evaluations of the *incumbent* configuration times a small bound multiplier.

ParamILS has led to very substantial improvements of SAT solvers for various applications. For example, it reduced the penalized average running time (with penalty factor 10, i.e., PAR-10)¹ of the SAT solver Spear [BH07] on a set of formal software verification instances by a factor of 500 [HBHH07]; this allowed Spear to win the QF_BV category of the SMT competition 2007. More recently, ParamILS was used to configure sat13, a high-performance SAT solver by Donald Knuth, achieving a 1.4-fold improvement in geometric average speedup over a default defined by the algorithm designer, even when tested on a diverse set of benchmark instances up to over three orders of magnitude harder than those in the given training set [Knu15]. This demonstrates that general-purpose automatic algorithm configuration can achieve very significant performance improvements

¹PAR-10 counts timeouts at a maximal captime $\bar{\kappa}$ as having taken time $10 \cdot \bar{\kappa}$ [HHLBS09].

over manual optimization carried out by the very best algorithm designers.

12.2.2. GGA and GGA++

The Gender-based Genetic Algorithm (GGA) [AST09] is a genetic algorithm for algorithm configuration that evolves a population of parameter configurations (having pairs of configurations mate and produce offspring) and uses the concept of gender to balance the diversity and fitness of the population. GGA's intensification mechanism for increasing the number of runs N it performs for each configuration over time keeps N constant in each generation, starting with small N_{start} in the first generation and linearly increasing N up to a larger N_{target} in a user-defined target generation and thereafter. GGA implements an adaptive capping mechanism by evaluating several candidate configurations in parallel until the first ones succeed and then terminating the remaining evaluations.

In experiments by Ans'otegui et al. [AST09], GGA performed more robustly than ParamILS on three configuration scenarios with 4 to 7 parameters. In a later comparison by Hutter et al. [HHLB11], ParamILS and GGA performed comparably across 15 configuration scenarios with 4 to 26 parameters, and ParamILS performed substantially better for the 76-dimensional configuration problem of optimizing the mixed integer programming solver CPLEX. GGA was also successfully used to configure solvers for various variants of MaxSAT, allowing the automated construction of portfolios that won several categories in the 2013 and 2014 MaxSAT evaluations [AMS14].

Inspired by the model-based approach underlying SMAC (see Section 12.2.3 below), Ans'otegui et al. [AMST15] introduced GGA⁺⁺, a version of GGA that relies on random forest models for performance prediction. More precisely, GGA⁺⁺ enhances the evolutionary approach underlying GGA by using a novel random forest variant in combination with a targeted sampling procedure to optimize configurations obtained by recombination. Ans'otegui et al. [ansotegui-ijcai15] reported promising results compared to GGA and SMAC on two algorithm configuration scenarios based on prominent SAT solvers; unfortunately, however, this comparison was confounded by two significant bugs in the algorithm wrapper (see Footnote 9 of Eggenberger et al. [ELH19] for details).

12.2.3. SMAC

Sequential, Model-based Algorithm Configuration (SMAC) [HHLB11] is an algorithm configuration method that uses a *empirical performance model* [LBNS09, HXHLB14] (that maps parameter values to algorithm performance and is learned from data) to identify algorithm configurations worth evaluating. In essence, SMAC can be understood as using this model to replace the local search component of ParamILS. SMAC begins by carrying out some initial algorithm runs on both the algorithm default and, optionally, a small set of random configurations. It calls the best performing of these the *incumbent* and then iterates the following steps until a given time budget is exhausted:

1. fit a probabilistic regression model to the performance data gathered so far;

2. use the model to determine a list of promising configurations², interleaving configurations sampled uniformly at random;
3. compare configurations from the promising list to the current incumbent, saving the resulting performance data, and updating the incumbent if appropriate.³

As with ParamILS, the mechanism used for comparing configurations is critical; SMAC uses an approach very similar to that of FocusedILS, performing adaptive capping and dedicating more running time to better-performing configurations. Another key element of SMAC involves the means by which the model is used to identify promising configurations; it uses a so-called *acquisition function* to trade off exploration (choosing configurations in parts of the space where the model has low confidence) with exploitation (choosing configurations in parts of the space where the model predicts fast running times). This acquisition function is computed based on model predictions and then optimised across the configuration space to identify the most promising next configuration.

SMAC can be used with different types of empirical performance models and different acquisition functions. Work using it to configure solvers for SAT and similar combinatorial problems most commonly uses random forests [Bre01] as its underlying performance model and adopts the expected improvement acquisition function [JSW98], optimised via iterated local search. Random forests are particularly suitable for modelling the running time of solvers for hard combinatorial problems, as they can easily handle mixed continuous/categorical inputs, have a low computational complexity for constructing the model and making predictions, act as automated feature selectors to detect the few most important inputs, and can deal with large heteroscedastic noise. However, since they do not usually make probabilistic predictions, they had to be extended by using randomly-chosen split points and uncertainties computed based on the discrepancy of the predictions across the different trees [HXHLB14].

Hutter et al. [HHLB11] used 17 different configuration scenarios to compare the performance of SMAC to that of GGA and ParamILS. In this evaluation, SMAC performed statistically significantly better than ParamILS in 11 of 17 cases (achieving speedup factors ranging from $0.93\times$ to $2.25\times$) and statistically significantly better than GGA in 13 of 17 scenarios (achieving speedup factors ranging from $1.01\times$ to $2.76\times$).

SMAC has also been used in numerous practical applications. Notably, it was used by the US Federal Communication Commission to build a solver for determining the feasibility of radio spectrum repackings in its 2016–17 Incentive Auction; this solver made use of SAT encodings and various prominent SAT algorithms. The stakes of this auction were extremely high: 84 MHz of radio spectrum were purchased across the whole US and Canada at a cost of over \$10 billion. As reported by Newman et al. [NFLB17], configuration with SMAC yielded large performance gains in this domain. Gnovelty+PCL [DP13] was the best single

²It is not necessary to identify more than one configuration in this step, but doing so can amortize the cost of fitting the model in Step 1.

³In order to ensure that the time for the construction and use of models does not outweigh the time for actually running algorithms, SMAC iteratively evaluates configurations from the promising list until the time for step 3 is larger than that required for steps 1 and 2 combined.

solver, able to solve 79.96% of representative problem instances within a minute; the parallel portfolio (see Section 12.4.3) of the best 20 algorithms could only solve 81.58% within the same amount of wall-clock time. After configuration, an eight-algorithm parallel portfolio based on SATenstein [KXHLB16] and clasp [GKNS07] identified using Hydra [XHLB10, see Section 12.4.1] was able to solve 96.03% of these problems in a minute (and 87.73% in only a second). Subsequent analysis showed that this improvement in solver performance is likely to have led directly to better economic outcomes, on the order of over \$700 million in cost savings and over \$300 million in additional economic value created by a more efficient auction outcome [NLBMS17].

12.2.4. The Configurable SAT Solver Challenges

The SAT community has a long history of regularly assessing the state of the art via competitions [JBR12]. The first SAT competition dates back to the year 2002 [SBH05], and the event has been growing over time: in 2018, as many as 106 solver versions participated across four tracks [HJS18].

A drawback of traditional solver competitions is that they fail to reward solver developers for identifying versatile, powerful designs that incorporate a large number of interchangeable heuristics; instead, they reward developers primarily for configuring their solvers in a way that ultimately targets the distribution used for evaluation. The *Configurable SAT Solver Challenge (CSSC)*, held alongside the SAT competition in 2013 and in 2014 as part of the 2014 FLoC Olympic Games, addressed this issue [HLB⁺17]. It evaluated SAT solver performance *after* application-specific customization, taking into account the fact that effective algorithm configuration procedures can automatically customize solvers for a given distribution of benchmark instances. Specifically, for each distribution \mathcal{D} of instances and each SAT solver S with configuration space Θ_S , an automated fixed-time offline configuration phase determined parameter settings of $\theta_S \in \Theta_S$ optimized for high performance on \mathcal{D} . Then, the performance of S on new instances from \mathcal{D} (not seen during the configuration process) was evaluated with these settings θ_S , and the solver with the best performance was declared the winner. To avoid bias arising from the choice of algorithm configuration method, the CSSC independently used all three of the most widely used algorithm configuration methods: ParamILS, GGA, and SMAC.

Overall, the two CSSC events held to date have yielded four main insights. First, algorithm configuration often improved performance substantially, in some cases yielding orders-of-magnitude speedups (e.g., in the 2014 CSSC, the PAR-10 score of clasp [GKNS07] improved from 705 to 5 on N -Rooks instances [MS14], and the PAR-10 score of ProbSAT [BS12] on 5-SAT instances improved from 3000 to 2, i.e., from solving no instance at all in 300 seconds to solving all instances in an average of 2 seconds). Second, some solvers benefited more from configuration than others, meaning that algorithm rankings were substantially different after configuration than before. For example, in the Random SAT+UNSAT track of CSSC 2013, the solvers ranking first (clasp [GKNS07]), second (Lingeling [Bie13]) and third (Riss3g [Man13]) would have ranked 6th, 4th and 5th, respectively, based on performance with default parameter settings. Third, the configuration

budget used also affected the ranking; in particular, algorithms with larger configuration spaces needed longer configuration budgets (on the order of days) to achieve peak performance. Fourth, out of the three configurators used in the CSSC 2014, SMAC yielded the best performance in 51 configuration scenarios, ParamILS in 17, and GGA in 4. If only SMAC had been used for the configuration process, the largest slowdown of any solver on test instances would have been by a factor of 1.5; the same slowdown factor would have been 1.8 if ParamILS had been used exclusively, and 30.5 if GGA had been used exclusively.

12.2.5. Other Approaches

Finally, we survey some recent algorithm configuration approaches whose focus is not on the strategy for selecting the next configuration to evaluate but on the resources used for each evaluation.

12.2.5.1. iRace

The iterative racing procedure *irace* [LIDLSB11] uses an intensification mechanism based on racing a set of candidate configurations against each other, i.e., performing runs for each of them on one instance at a time and dropping candidates whose performance is statistically dominated. Based on the evaluations thus gathered, it constructs one-dimensional density estimates of which values perform well for each parameter; its search strategy is then an estimation of distribution approach that samples individual parameter values from these one-dimensional distributions. To increasingly focus the sampling process towards the most promising configurations, *irace* gradually decreases the standard deviations of these distributions (this process is dubbed *volume reduction*). Recently, Pérez Cáceres et al. [PCLHS17] have integrated a capping mechanism inspired by that used in ParamILS into *irace* and reported promising initial results on various configuration scenarios.

12.2.5.2. Structured Procrastination

The *Structured Procrastination (SP)* algorithm [KLBL17] is notable for offering the first nontrivial performance guarantees for general algorithm configuration with an average running time minimization objective. This work considered a worst-case setting in which an adversary causes every deterministic choice to play out as poorly as possible, but where observations of random variables are unbiased samples. In this setting, it is straightforward to argue that any fixed, deterministic heuristic for searching the space of configurations can be extremely unhelpful. The approach underlying SP therefore focuses on obtaining candidate configurations via random sampling.

Any method based on random sampling will eventually encounter the optimal configuration; the crucial question is the amount of time that this will take and whether the algorithm is guaranteed to eventually recognize the optimal configuration as such when it has been sampled often enough. Out of the methods surveyed so far, in discrete configuration spaces, ParamILS and SMAC provably converge to the optimal configuration, while GGA and GGA++ do not have this

property.⁴ The key result of Kleinberg et al. [KLBL17] is that SP is guaranteed to find a near-optimal configuration with high probability, with worst-case running time that nearly matches a lower bound on what is possible and that asymptotically dominates that of existing alternatives, such as SMAC.

Unfortunately, SP is not useful in practice; it takes an extremely long time to run, even on configuration scenarios that are easy for existing methods, and hence has not been implemented or evaluated empirically. The problem is that SP treats every instance like the worst case, in which it is necessary to achieve a fine-grained understanding of every configuration’s running time in order to distinguish between them. Extending SP to make it practical remains an active topic of research. Notably, a new variant called *Structured Procrastination with Confidence* [KLBLG19] maintains SP’s theoretical guarantees while improving performance outside the worst case. While it is more promising than SP, this very recent algorithm has only been evaluated empirically on a single dataset to compare it to the two algorithms discussed next. (In this comparison, it achieved better performance, but a much more extensive study would be required to draw reliable conclusions.)

12.2.5.3. LeapsAndBounds, CapsAndRuns

A configuration procedure called LeapsAndBounds [WGS18b] is based on SP but improves upon it in various ways. In theoretical terms, LeapsAndBounds offers an improvement by more closely matching the lower bound in SP’s analysis and refining that analysis to remove dependence on one of the parameters used. More importantly for our discussion here, LeapsAndBounds also improves performance outside the worst case. A second refinement of SP called CapsAndRuns [WGS18a] goes even further in this direction. Limited initial experiments (running Minisat [ES04] on SAT instances generated using CNFuzzDD (<http://fmv.jku.at/cnfuzzdd>) show considerable promise over the original version of SP. However, LeapsAndBounds has not been evaluated beyond this single setting, and has not been compared to ParamILS, SMAC, GGA/GGA++ or irace.

12.3. Per-instance algorithm selection

Per-instance algorithm selection entails automatically choosing from a “portfolio” of different algorithms for a given problem (such as SAT) every time a new problem instance is presented [Ric76]. This choice is made based on features of the given instance π , as illustrated in Figure 12.2. Like an algorithm configurator, a per-instance algorithm selector is a meta-algorithm that takes a set of algorithms as part of its input.

More formally, the *per-instance algorithm selection problem* is defined as follows. We start with a set \mathcal{P} of algorithms, which we call a *portfolio*, a distribution \mathcal{D} over possible inputs with domain Π , and a cost metric $c : \mathcal{P} \times \Pi \rightarrow \mathbb{R}$. We will

⁴GGA and GGA++ use a finite maximum number N_{end} to evaluate each configuration, but indefinite growth of this number is required for reliably selecting the optimal configuration in the presence of noise.

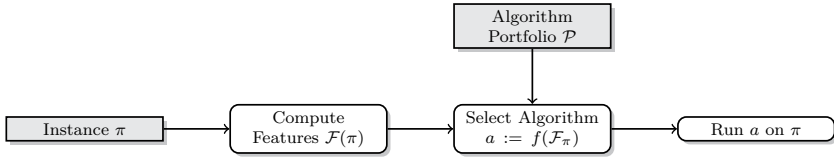


Figure 12.2. The computational core of algorithm selection at test time: for each new instance π , features $\mathcal{F}(\pi)$ are computed, and based on these, function f selects an algorithm a from the set of candidate algorithms \mathcal{P} to run on π .

summarize problem instances as vectors of ϕ real-valued *features*; we represent this reduction as $\mathcal{F} : \mathcal{D} \rightarrow \mathbb{R}^\phi$. Our goal is to construct a function $f : \mathbb{R}^\phi \rightarrow \mathcal{P}$ that maps any given input (here: a SAT instance) to an algorithm from the portfolio; this function is called a *per-instance algorithm selector*. Typically, the aim is to minimize c in expectation over \mathcal{D} , but more complex goals are possible, such as aiming for diversity across subcategories of problem instances; indeed, such evaluation functions have been used in past SAT competitions.

Per-instance algorithm selection has two distinct phases: an offline training phase, in which a selector is constructed, and an online use phase, where that selector is used to determine the algorithm to be run on new problem instances π . As we will see in the following, selector construction is often approached as a supervised machine learning task, using (cost-based) classification or regression techniques, although there also exist approaches based on feature-based clustering of instances.

In the following, we consider per-instance algorithm selection approaches along with two extensions, which turn out to be important for achieving state-of-the-art performance in SAT and other problems: pre-solving schedules and backup solvers. The former are sequences of algorithms run prior to the algorithm selection process, mostly in order to avoid potentially costly feature extraction on instances that are solved easily; the latter are used as a fall-back when feature computation fails. We describe in the greatest detail approaches that have had broad impact in the SAT community and that have been demonstrated to achieve both strong and robust performance.

12.3.1. SATzilla

The first version of SATzilla was built for the 2003 SAT Competition and was indeed the first algorithm selection system to be entered into such a competition [NLBA⁺03, NLBH⁺04]. It used independent ridge regression models to predict the performance of component SAT solvers based on a rich set of up to 84 instance features, and selected the solver with the best predicted running time. These models were trained in an offline phase on a broad and diverse set of training instances. In the 2004 submission, a local-search-based pre-solver was added to quickly solve easily satisfiable instances without incurring the cost of computing features. Overall, SATzilla won two silver and one bronze medal across the nine tracks of the 2003 SAT Competition and two bronze medals in the 2004

SAT Competition, providing an early indication for the potential of per-instance algorithm selection for SAT.

For the 2007 SAT Competition, this basic design was elaborated in various ways [XHHLB07]. Two key updates were made to the empirical performance models. First, SATzilla 2007 was based on so-called hierarchical hardness models, using a sparse multinomial logistic regression (SMLR) classifier to predict the probability of instance satisfiability and then using this result to weight the predictions of two quadratic regression models, trained only on satisfiable and unsatisfiable instances respectively [XHLB07]. Second, a method from survival analysis [SH79] was used to address bias stemming from the fact that true algorithm running times are not fully observed when long runs are capped. Several other innovations tuned performance by optimizing on a separate validation set of instances. A pre-solver selection mechanism allowed up to two solvers to be run sequentially before feature computation; an automatic solver subset selection mechanism was used to drop solvers whose inclusion worsened overall performance; and a backup solver was identified for use when feature computation failed. Finally, the set of features was somewhat reduced (to 48) to exclude those whose time complexity exceeded their marginal value. This version of SATzilla was very successful in the 2007 SAT Competition, winning three gold, one silver, and one bronze medal. Following the 2007 competition, SATzilla was improved further by making the selector construction more scalable and completely automated, by integrating candidate solvers based on local search, by predicting competition score instead of running time, and by using hierarchical hardness models that take into account different types of SAT instances [XHHLB08]. The resulting version won three gold and two silver medals in the 2009 competition [XHHLB09].

In 2011, a substantially improved version of SATzilla was introduced, which showed outstanding performance in the evaluation track of the 2011 SAT Competition and in the 2012 SAT Challenge [XHHLB12]. This version of SATzilla differed from the previous design by using cost-sensitive decision forests to predict for every pair of component solvers the one that would perform better on a given SAT instance, then employing majority voting over these predictions to select the solver to run. Specifically, the pairwise classification mechanism at the heart of this approach penalized misclassifications in direct proportion to their impact on overall selector performance, without predicting running time or competition score. As in earlier versions of SATzilla, pre- and backup-solvers were used in addition to the main per-instance algorithm selection mechanism. However, the decision about whether to initiate feature computation (rather than immediately running the statically determined backup solver) was made by a random forest classifier, using a subset of cheaply computable features. In the 2012 SAT Competition, this version of SATzilla used a total of 31 SAT solvers and 138 instance features; it ended up winning first prizes in two of the three main tracks and second prize in the remaining track, while also placing first in the sequential portfolio track. In the application track of this competition, it solved 6.4% more instances than the single best “interacting multi-engine” solver (which combined multiple SAT solving engines in a design that allows for richer interaction than found in a per-instance algorithm selector like SATzilla), 11.8% more instances than the

best conventional, “single-engine” solver, and only 6.5% fewer instances than the virtual best solver (a hypothetical, perfect selector).

12.3.2. ISAC, 3S and CSHC

ISAC (Instance-Specific Algorithm Configuration) uses a clustering approach to perform algorithm selection [KMST10]. Specifically, ISAC clusters the instances in the given training set based on their features, using the g -means algorithm in combination with a post-processing scheme that reassigns instances from clusters whose size falls below a user-defined threshold. For each cluster, the cluster centre and the best-performing solver (from the given set of component solvers) are stored; given a new problem instance, ISAC determines the nearest cluster centre (using a 1-nearest neighbour approach) and runs the solver associated with it. ISAC was originally designed to combine algorithm configuration and per-instance selection, by automatically configuring a single, parametric solver for each cluster (see Section 12.4.1). Initial results for SAT and two other problems (set covering and mixed integer programming) were promising, but limited in scope; in the case of SAT, ISAC was applied to a single local-search-based SAT solver, SAPS [HTH02], on which it showed substantial speedups over the default configuration on several benchmarks [KMST10]. Later, used as a pure per-instance algorithm selector in combination with a large number of high-performance SAT solvers, ISAC was demonstrated to achieve substantial performance improvements compared to the single best solver on a broad set of random SAT instances from several SAT competitions, and more modest improvements on hand-crafted and industrial SAT instances; however, in all cases, performance in terms of PAR-10 scores was more than a factor of 9.9 worse than that of the virtual best solver [CMMO13].

The clustering approach used by ISAC also underlies 3S (semi-static solver schedules), which combines per-instance algorithm selection with sequential algorithm scheduling [KMS⁺11]. Like ISAC, 3S uses the g -means algorithm for clustering problem instances. For a given instance i to be solved, it then uses a k -nearest neighbour approach to determine a set of similar training instances and selects the algorithm with the best performance on this instance set. This mechanism is enhanced by weighting algorithm performance on a reference instance i' by the distance of i' to i , and by adapting the neighbourhood size k based on the features of i . 3S incorporates the per-instance algorithm selector thus obtained into a static algorithm schedule, determined using mixed integer programming. While this approach is conceptually similar to the pre-solver schedules used in SATzilla from 2007 onwards, it offers considerably more flexibility in the design of sequential schedules. On random SAT instances, 3S was reported to reduce the gap between the 2009 version of SATzilla (which then constituted the state of the art for these types of instances) and the virtual best solver (VBS) (a hypothetical, perfect per-instance algorithm selector) by 57%. Unlike SATzilla, 3S participated in the 2011 SAT Competition, where it showed impressive performance, winning 2 gold, 2 silver and 3 bronze medals. In a comparison based on scenarios from the 2012 SAT Challenge, 3S was reported to obtain PAR-10 scores between 49% and 81% worse than those reached by the 2012 version of SATzilla [MSSS13].

Like ISAC and 3S, the CSHC (cost-sensitive hierarchical clustering) algorithm selection system is based on the idea of feature-based partitioning of problem instance space into clusters [MSS13]. However, instead of unsupervised clustering, as used in ISAC and 3S, CSHC creates these clusters using a supervised learning algorithm. Effectively, it thus constitutes a cost-sensitive approach for constructing decision trees (we note that all decision tree algorithms determine hierarchical rules for “clustering” the training examples stored in each leaf node). The reader may recall that more recent versions of SATzilla also relied on (random forests of) cost-sensitive decision trees; however, the approaches are different. SATzilla uses decision trees to predict which of two algorithms will be faster, followed by voting across quadratically many such classifiers to choose a single algorithm; CHSC’s decision trees directly predict the best algorithm to run. More specifically, starting with all instances in the given training set, CSHC recursively partitions a given “cluster” of instances into two “sub-clusters”, based on a single feature value, such that the performance of the best-performing algorithm in each “sub-cluster” is optimized. When asked to solve a new instance i , CSHC first runs the 3S static algorithm schedule for a fixed fraction (10%) of the overall time budget. If this does not solve the instance, it computes features and uses the learned decision tree to identify a leaf node and thereby the algorithm that performed best on that “cluster” of instances. CSHC was reported to achieve slightly better performance than the 2012 version of SATzilla on a number of scenarios based on the 2012 SAT Challenge, although the observed performance differences were reported not to be statistically significant [MSS13].

12.3.3. SNNAP

A combination of supervised and unsupervised learning forms the basis for SNNAP (solver-based nearest neighbour for algorithm portfolios) [CMMO13]. Specifically, SNNAP uses random forests to predict the running times of individual solvers; these predictions then provide the basis for identifying instances from the given training set that are similar to the instance to be solved. Instance similarity is assessed based on a (small) fixed number of best solvers known or predicted to perform best on a given instance, using Jaccard distance. The solver to be run is then selected as the one predicted to perform best on the k nearest instances in the given training set to the instance to be solved, for a user-defined value of k . In the experiments reported by Collautti et al., $k = 60$ and similarity calculations are based on the three best solvers for any given instance.

SNNAP has been demonstrated to show substantially stronger performance than ISAC on large collections of instances collected from SAT competitions between 2006 and 2012 [CMMO13]. A comparison against a 2015 internal development version of SATzilla, which is a minor variant of the 2011 version described previously, revealed that on 5 of 7 scenarios based on the 2011 SAT Competition and the 2012 SAT Challenge, SNNAP performed significantly worse than SATzilla, with statistically tied and slightly better performance, respectively, on the two remaining scenarios [LHHS15].

12.3.4. AutoFolio

As is evident from the previous sections, when designing an algorithm selection system, many design choices have to be made. These choices can have substantial impact on performance, and what works best generally depends on the given use case. Therefore, following the previously outlined programming by optimization approach, AutoFolio combines a highly parametric framework for algorithm selection with a general-purpose algorithm configurator, which is used to instantiate this framework in such a way that performance is optimized for a given use case (such as a specific set or distribution of SAT instances [LHHS15]). AutoFolio is not restricted to SAT, but can be applied to arbitrary algorithm selection problems.

The underlying algorithm selection framework, called *claspfolio 2*, was originally developed in the context of algorithm selection for answer set programming (ASP) and incorporates techniques and ideas from a broad range of high-performance per-instance algorithm selection approaches [HLS14]. This framework encompasses strategies from a broad range of algorithm selection systems, including 3S [KMS⁺11], ISAC [KMST10], SATzilla [XHHLB08, XHHLB12], and SNNAP [CMMO13]; for each selection approach, AutoFolio considers subsidiary design choices and parameter settings. Furthermore, several techniques for pre-processing the training data from a given algorithm selection scenario and for constructing pre-solver schedules can be configured independently from the selection approach. Automated configuration of this flexible framework is performed using SMAC [HHLB11], in combination with cross-validation on the given training set to robustly evaluate the performance of candidate configurations.

AutoFolio has been empirically evaluated against other algorithm selection techniques across a broad range of algorithm selection scenarios, including several benchmark sets derived from SAT competitions [LHHS15]. In those experiments, AutoFolio was found to be the only approach that achieved state-of-the-art performance for all benchmarks; notably, for each benchmark scenario, it was statistically tied with or exceeded the performance of the best of SNNAP, ISAC and SATzilla. However, there remained a considerable gap to the performance bounds obtained from the virtual best solvers for these algorithm selection scenarios (e.g., a factor of 8.8 in PAR-10 score between the AutoFolio selector and the VBS, compared to a factor of 1.8 between the single best solver and the AutoFolio selector on the SAT12-INDU benchmark), indicating room for further improvements in automated algorithm selection.

12.3.5. The Sparkle SAT Challenge

Solver competitions, such as the quasi-annual SAT solver competitions and races, serve several purposes. Aside from providing an objective, neutral performance assessment of the participating algorithms, they also provide a powerful incentive for improving the state of the art in solving the problem in question. Unfortunately, as discussed in Section 12.2.4, traditional competition designs assess – and incentivize improvement of – broad-spectrum performance of fairly monolithic solvers. This becomes an issue for problems such as SAT, where the overall

state of the art is represented by a set of algorithms, and more precisely for algorithm selectors, since they leverage the performance complementarity in such a solver set, rather than simply benefiting from the performance of the fastest single algorithm.

Therefore, just as the Configurable SAT Solver Challenges built the existence of powerful, general-purpose algorithm configurators into the design of a competition, the Sparkle SAT Challenge is predicated on the ready availability of per-instance algorithm selection systems for SAT. The key idea underlying this competition, which was held as part of the 2018 FLoC Olympic Games, was to let competitors submit SAT solvers that would be integrated into a per-instance algorithm selector constructed using AutoFolio [LHHS15]. Participating solvers would then be evaluated based on the degree to which they contribute to the overall performance of the selection system thus obtained [LH18]. To make it easier for participants to optimize the contributions of their solvers, and hence not only their standing in the competition, but also the overall performance of the final algorithm selector, daily evaluations (on a set of training instances) were conducted and published during a leader-board phase.

The Sparkle SAT Challenge was conducted using a newly designed software platform called Sparkle, which aims to make the construction and evaluation of algorithm selectors more easily accessible to solver developers and practitioners. It used the well-known feature extractor by Xu et al. [XHHLB12] in combination with the state-of-the-art AutoFolio selector construction system discussed in Section 12.3.4. Solver contributions to overall selector performance were assessed using relative marginal contribution (RMC), a normalised version of marginal contribution [XHHLB12]. More sophisticated approaches exist, but would have been computationally too expensive to use in the competition [see, e.g., FKM⁺16, FKM⁺18]. While official results were based on RMC to an actual selector obtained from AutoFolio, RMC to the virtual best solver over the same set of component algorithms was also reported. It is worth noting that the Sparkle system used to run the challenge not only fully automates the construction of the selector, but also the comprehensive evaluation of component solver and selector performance, which is compiled into a detailed technical report generated by Sparkle.

The results from the Sparkle SAT Challenge, as presented during the 2018 FLoC Olympic Games [LH18], gave rise to three main findings.

First, as observed consistently in the literature and in other competitions, using per-instance algorithm selection, performance complementarity between SAT solvers was leveraged, leading to substantial performance improvements over the single best solver (SBS). These improvements tended to be much larger than those between the SBS and the runners-up in terms of stand-alone solver performance. The difference in stand-alone performance between the SBS and the runner-up was less than 1% in terms of PAR2 score, while the automatically constructed selector was more than 7.5% better than the SBS, despite the fact that the test instances differed substantially from the training instances used as a basis for constructing the selector. This highlights the continued importance of per-instance algorithm selection for achieving state-of-the-art performance in solving SAT.

Second (as previously observed by Xu et al. [XHHLB12]) the stand-alone per-

formance of SAT solvers tended to be uncorrelated with their contributions to a state-of-the-art per-instance selector built from them, and likewise uncorrelated with their contributions to the VBS. This suggests that conventional competitions may not be the most effective way of improving the state of the art in SAT solving, since they motivate solver developers to optimize stand-alone performance. In the Sparkle SAT Challenge, the solver ranked third in terms of stand-alone performance had a relative marginal contribution to selector performance that was over five times lower than that of the solver ranked last.

Third, when given the opportunity to repeatedly evaluate the contributions of their solvers to overall selector performance, solver developers appear to have succeeded in maximizing those contributions and, as a result, also overall selector performance. We attribute this partly to the easy and frictionless access to cutting-edge selector construction methods offered by the Sparkle platform. In particular, selector performance on training data accessible to participants in the challenge improved by a factor of more than 2 in terms of PAR2 score.

12.4. Related approaches

Automated algorithm configuration and per-instance algorithm selection, although very widely studied in the literature, are not the only meta-algorithmic design techniques. In this section, we briefly outline additional techniques, many of which have been applied to SAT.

12.4.1. Combining algorithm configuration and algorithm selection

Algorithm configuration and algorithm selection have different strengths. Algorithm configuration is good at searching enormous parameter spaces to find settings that yield peak average performance, at a cost of hours or days of off-line compute time. Algorithm selection is good at exploiting variability in the running times of different solvers (or different configurations) *within* instance distributions of interest, and can select the best of a small, fixed set of solvers quickly enough to be used online. In many practical applications, the algorithm configuration setting is closer to the needs of a typical practitioner than the algorithm selection setting: notably, it can be much easier to maintain a single algorithm codebase with parameterized components than different codebases for multiple solvers. However, sometimes different codebases arise separately (as in the SAT competition setting), and sometimes practitioners face heterogeneous sets of problem instances, on which no single configuration yields sufficiently strong average performance. In order to get the best of both worlds, we can use algorithm configuration to obtain a set of complementary configurations that can be fruitfully combined using algorithm selection.

One simple approach to combining algorithm configuration and algorithm selection is to cluster a heterogeneous set of problem instances into a set of homogeneous subsets, to run configuration on each of these subsets to obtain specialized configurations, and then to treat these specialized configurations as the solvers to be combined with a per-instance algorithm selector. This is the approach followed by *instance-specific algorithm configuration* (ISAC [KMST10]).

A more sophisticated way to combine configuration and selection is to use configuration to iteratively identify configurations that are maximally complementary to an existing set. *Hydra* [XHLB10] follows this idea, building on an approach first proposed by Leyton-Brown et al. [LBNA⁺03]. *Hydra* starts from an empty portfolio and then iteratively adds a new configuration that maximally improves the oracle performance of the configurations in the portfolio. Because the oracle performance of an algorithm portfolio is a submodular function, this approach is guaranteed to yield portfolios whose oracle performance is within a factor of $1 - 1/e$ from optimal.

12.4.2. Algorithm schedules

While per-instance algorithm selection often works very well, it depends on the availability of cheaply computable instance features that jointly correlate well with algorithm running time. An alternative strategy that does not depend on the existence of such features is building an *algorithm schedule* that runs several algorithms in a fixed or adaptive sequence. Algorithm schedules work when many solvers have high variance in their running time distributions across instances (i.e., solve some instances very quickly and others very slowly) and when different instances are easy for different solvers.

Indeed, algorithm schedules can also be useful even when informative instance features do exist, because they offer a way to solve easy instances without computing features. This matters particularly when performance is measured by making a multiplicative rather than additive comparison to optimal, because feature computation can dominate running time on easy instances. It is for this reason that several per-instance algorithm selection systems use pre-solver schedules, as described in Section 12.3.

Algorithm schedules can be constructed using various techniques. They can be defined by hand, built by local search in the joint space of algorithms and running times (as done for determining the pre-solving schedules used by SATzilla [XHHLB08]), or by means of mixed integer programming [KMS⁺11] or answer set programming [HKLS15]. It is also possible to start with an empty schedule and use a *Hydra*-like approach to greedily append one configuration at a time to the current schedule, along with a time budget for which it should be allowed to run [SSH15]. (In contrast to *Hydra*, the objective function optimized by algorithm configuration in this case is performance gain of the schedule *per additional time spent*, but otherwise the approach is similar.)

12.4.3. Parallel portfolios

So far, this chapter has focused on approaches that require only a single processor core. Of course, modern computing is increasingly parallel, particularly as cloud resources become available. Parallel portfolios are based on the idea of running multiple sequential solvers in parallel on a given problem instance to be solved; if these component solvers have complementary strength, this can lead to an improvement in performance in terms of wall-clock time and, in some cases, even in terms of overall CPU time, when solving decision problems such as SAT.

Parallel portfolios can thus be used to construct solvers that effectively exploit parallel computing resources without the well-known difficulties associated with developing intrinsically parallel solvers.

Even simple parallel portfolios can be very powerful. To give one notable example, in the 2011 SAT Competition, the solver pfolio [Rou11] won 11 medals in the parallel track and 5 medals in the sequential track (where its parallelism was realised by task switching) by following an incredibly straightforward strategy: running the best solvers from several categories of the 2009 SAT Competition in parallel. This approach was effective, because the solvers forming the parallel portfolio performed well individually and complemented each other well. Roughly speaking, automated methods for constructing parallel portfolios also aim to identify an optimal compromise between complementarity and individual strength.

Algorithm configuration and per-instance selection can both be extended to the parallel case. Algorithm configuration can be used to create strong parallel solvers based only on the parametric code of a sequential solver [LHLBS17]. Once more, Hydra's greedy approach is effective: starting with an empty portfolio (on zero processor cores), greedily add the configuration to run on the next processor core that most improves performance when run alongside the existing portfolio. Algorithm selection can also be generalized to produce a parallel portfolio of solvers rather than a single solver [LHH15]. In particular, a rather broadly applicable approach for parallelizing an existing algorithm selection approach is to adapt its underlying machine learning model to rank the solvers, and then simply to select the top-ranked K solvers to concurrently run on K processor cores.

12.4.4. Parameter control

Just as different solvers and solver configurations exhibit strong performance on different problem instances, so it is the case that different strategies work best during different phases of solving a single instance. For example, local search algorithms need to effectively switch between exploration and exploitation phases, and tree search algorithms may benefit from using different heuristics at different depths of the search tree or as a function of the number of clauses already learned. This observation gives rise to the meta-problem of adapting the parameters of a given algorithm online as a function of the current search state. While algorithm configuration can be formulated as a black-box optimization problem and algorithm selection as a supervised learning problem, *parameter control* is a more complex problem akin to reinforcement learning [LL00, LL01, EFW⁺02, BC12, AN16, BBHL19].

12.4.5. Parameter importance analysis

It is important to complement the automated methods for constructing strong solvers discussed up to this point with analysis techniques that can yield insights about what components were responsible for observed performance improvements. For algorithm configuration, it is possible to use model-based approaches to detect which parameters have an important influence on performance. Techniques

can be based on forward selection [LBNS09, HHLB13]; based on a decomposition of the variance of predicted performance into components attributable to each single parameter and combination of (two or more) parameters [HHLB14]; or based on local effects around the best found configuration [BMLH18]. Alternatively, ablation analysis can be used to assess the relative contributions of individual parameters to observed performance differences between algorithm configurations [FH16]. It is also possible to perform a model-based variant of such an ablation analysis, which means that the analysis can be performed without executing additional runs of the algorithm under study [BLE⁺17].

12.4.6. Using algorithm configuration to find bugs in solvers

As mentioned at the beginning of this chapter, algorithm configuration has uses beyond performance optimization; one of these is the identification of parameter configurations that lead to errors, and especially those that do so quickly. Algorithm configuration optimizes a user-defined performance metric; in SAT solving, common choices are minimizing the number of timeouts for a given cutoff time and minimizing (penalized) average running time. A subtlety is how one should count runs that terminate quickly but either with no result (e.g., due to a segmentation fault) or with an incorrect result (e.g., UNSAT for a satisfiable instance). When seeking to produce good average performance, such runs are typically given a terrible score (e.g., the timeout multiplied by a large constant) in order to drive the configurator towards areas of the configuration space that do not cause erroneous behaviour. In contrast, blindly minimizing average measured running time actually causes the configurator to find configurations that terminate as quickly as possible; often this yields buggy configurations that *break* quickly [ELH19]. This can be viewed as a tool for automatically finding whitebox tests that expose problems quickly [HHLB10, ML16].

12.5. Conclusions and open challenges

The meta-algorithmic techniques described here have already improved the state of the art in SAT solving for various types of applications [see, e.g., HBHH07, NFLB17], but there still remain several open challenges in terms of methods, tools and applications.

Regarding methods, one of the biggest challenges is to develop an approach that works robustly, even for SAT instances that do not resemble the instances seen at training time. It would be highly desirable to have available an effective configuration method for highly heterogeneous instances, which could be used to identify better default configurations. However, in many cases, there simply does not exist a single configuration that performs well across instances. Instead, we believe that it might be more promising to develop a robust approach for selecting the best out of a small number of automatically-determined default configurations. An interesting research challenge consists of identifying such robust defaults based on a limited number of training instances. Ways to effectively integrate domain knowledge from solver developers would be extremely helpful towards this goal.

In terms of tools, it would be helpful to make it easier for developers and users of SAT solvers to take advantage of meta-algorithmic tools and the performance improvements they afford. Currently, considerable expertise is required to determine, e.g., which parameters should be configured for which kinds of instance sets, and to automatically build robust solvers for application to new instances, using approaches such as Hydra. This high barrier to entry could be lowered, at least in part, by designing a highly parametric CDCL solver, with a rich configuration space that covers most of the techniques used in modern CDCL-based SAT solvers, similar to the way that SATenstein [KXHLB09, KXHLB16] covers a substantial part of the space of stochastic local search solvers. Furthermore, there remains a need for making automatic algorithm configuration and per-instance selection methods easier to use for practitioners.

We see significant room for further innovation in solver competitions, which have historically been a significant driver of new work in the field. We see substantial value in competitions that deeply integrate both automated algorithm configuration (as done in the Configurable SAT Solver Challenges) and per-instance algorithm selection (as done in the Sparkle SAT Challenge). We are also excited about recent work that has shown how ideas from coalitional game theory can be used to derive new evaluation metrics for such competitions [FKM⁺16, KFM⁺18]. We hope that these metrics can be used in future competitions (perhaps alongside existing alternatives) to further improve the way in which innovative ideas in solver design are recognized.

Finally, as is evident both from the literature and from competitions, algorithm configuration and algorithm selection have the potential to produce performance gains in solving virtually any application problem. Specifically, we see significant opportunities in applying meta-algorithmic technologies to improve SMT solvers, which play a key role in some of the most important applications of SAT-related techniques in the areas of hardware and software verification.

References

- [AMS14] C. Ansótegui, Y. Malitsky, and M. Sellmann. Maxsat by improved instance-specific algorithm configuration. In *Proceedings of the Twenty-eighth National Conference on Artificial Intelligence (AAAI'14)*, pages 2594–2600. AAAI Press, 2014.
- [AMST15] C. Ansótegui, Y. Malitsky, M. Sellmann, and K. Tierney. Model-based genetic algorithms for algorithm configuration. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI'15)*, pages 733–739, 2015.
- [AN16] S. Adriaensen and A. Nowé. Towards a white box approach to automated algorithm design. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI'16)*, pages 554–560. AAAI Press, 2016.
- [AST09] C. Ansótegui, M. Sellmann, and K. Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In *Proceedings of the Fifteenth International Conference on Principles and Practice of Constraint Programming (CP'09)*, volume 5732 of

- Lecture Notes in Computer Science*, pages 142–157. Springer Verlag, 2009.
- [BBBK11] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In *Proceedings of the 25th International Conference on Advances in Neural Information Processing Systems (NIPS'11)*, pages 2546–2554, 2011.
- [BBHL19] A. Biedenkapp, H. F. Bozkurt, F. Hutter, and M. Lindauer. Towards white-box benchmarks for algorithm control. In *IJCAI 2019 DSO Workshop*, August 2019.
- [BC12] R. Battiti and P. Campigotto. An investigation of reinforcement learning for reactive search optimization. In Youssef Hamadi, Eric Monfroy, and Frédéric Saubion, editors, *Autonomous Search*, pages 131–160. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
- [BH07] D. Babić and F. Hutter. Spear theorem prover. Solver description, SAT competition 2007, 2007.
- [Bie13] A. Biere. Lingeling, Plingeling and Treengeling entering the SAT competition 2013. In *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of *Department of Computer Science Series of Publications B*, pages 51–52. University of Helsinki, 2013.
- [BLE⁺17] A. Biedenkapp, M. Lindauer, K. Eggenesperger, C. Fawcett, H. Hoos, and F. Hutter. Efficient parameter importance analysis via ablation with surrogates. In *Proceedings of the Conference on Artificial Intelligence (AAAI'17)*, pages 773–779. AAAI Press, 2017.
- [BMLH18] A. Biedenkapp, J. Marben, M. Lindauer, and F. Hutter. Cave: Configuration assessment, visualization and evaluation. In *Proceedings of the International Conference on Learning and Intelligent Optimization (LION'18)*, 2018.
- [Bre01] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.
- [BS12] A. Balint and U. Schöning. Choosing probability distributions for stochastic local search and the role of make versus break. In *Proceedings of the Fifteenth International Conference on Theory and Applications of Satisfiability Testing (SAT'12)*, volume 7317 of *Lecture Notes in Computer Science*, pages 16–29. Springer Verlag, 2012.
- [BYC14] J. Bergstra, D. Yamins, and D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *Proceedings of the 30th International Conference on Machine Learning (ICML'13)*, pages 115–123. Omnipress, 2014.
- [CFH08] M. Chiarandini, Chris Fawcett, and H. Hoos. A modular multiphase heuristic solver for post enrolment course timetabling. In *Proceedings of the Seventh International Conference on the Practice and Theory of Automated Timetabling*, 2008.
- [CMMO13] M. Collautti, Y. Malitsky, D. Mehta, and B. O'Sullivan. SNNAP: Solver-based nearest neighbor for algorithm portfolios. In *Machine Learning and Knowledge Discovery in Databases (ECML/PKDD'13)*, volume 8190 of *Lecture Notes in Computer Sci-*

- ence, pages 435–450. Springer Verlag, 2013.
- [DP13] T.-T. Duong and D.-N. Pham. gNovelty+GC: Weight-Enhanced Diversification on Stochastic Local Search for SAT. In *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of *Department of Computer Science Series of Publications B*, pages 49–50. University of Helsinki, 2013.
- [DSH15] T. Domhan, J. T. Springenberg, and F. Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI’15)*, pages 3460–3468, 2015.
- [EFW⁺02] S.L. Epstein, E.C. Freuder, R. Wallace, A. Morozov, and B. Samuels. The adaptive constraint engine. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP’02)*, pages 525–542, London, UK, UK, 2002. Springer-Verlag.
- [ELH19] K. Eggenesperger, M. Lindauer, and F. Hutter. Pitfalls and best practices in algorithm configuration. *Journal of Artificial Intelligence Research*, 64:861–893, 2019.
- [ES04] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proceedings of the conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer Verlag, 2004.
- [FH16] C. Fawcett and H.H. Hoos. Analysing differences between algorithm configurations through ablation. *Journal of Heuristics*, 22(4):431–458, 2016.
- [FKE⁺15] M. Feurer, A. Klein, K. Eggenesperger, J. T. Springenberg, M. Blum, and F. Hutter. Efficient and robust automated machine learning. In *Proceedings of the 29th International Conference on Advances in Neural Information Processing Systems (NIPS’15)*, 2015.
- [FKM⁺16] A. Fréchette, L. Kotthoff, T.P. Michalak, T. Rahwan, H. Hoos, and K. Leyton-Brown. Using the shapley value to analyze algorithm portfolios. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI-16)*, pages 3397–3403. AAAI Press, 2016.
- [GKNS07] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. clasp: A conflict-driven answer set solver. In *Logic Programming and Non-monotonic Reasoning*, pages 260–265. 2007.
- [HBHH07] F. Hutter, D. Babić, H. Hoos, and A. Hu. Boosting verification by automatic tuning of decision procedures. In *Formal Methods in Computer Aided Design (FMCAD’07)*, pages 27–34, 2007.
- [HHLB10] F. Hutter, H. Hoos, and K. Leyton-Brown. Automated configuration of mixed integer programming solvers. In *Proceedings of the Seventh International Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR’10)*, volume 6140 of *Lecture Notes in Computer Science*, pages 186–202. Springer Verlag, 2010.
- [HHLB11] F. Hutter, H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proceedings of*

the Fifth International Conference on Learning and Intelligent Optimization (LION'11), volume 6683 of *Lecture Notes in Computer Science*, pages 507–523. Springer Verlag, 2011.

- [HHLB13] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Identifying key algorithm parameters and instance features using forward selection. In *Proceedings of the 7th International Conference on Learning and Intelligent Optimization (LION-7)*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, January 2013.
- [HHLB14] F. Hutter, H. Hoos, and K. Leyton-Brown. An efficient approach for assessing hyperparameter importance. In *Proceedings of International Conference on Machine Learning 2014 (ICML'14)*, pages 754–762, June 2014.
- [HHLBS09] F. Hutter, H. Hoos, K. Leyton-Brown, and T. Stützle. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.
- [HHS07] F. Hutter, H. Hoos, and T. Stützle. Automatic algorithm configuration based on local search. In *Proceedings of the Twenty-second National Conference on Artificial Intelligence (AAAI'07)*, pages 1152–1157. AAAI Press, 2007.
- [HJS18] M. Heule, M. Järvisalo, and M. Suda, editors. *Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions*, volume B-2018-1 of *Department of Computer Science Series of Publications B*. University of Helsinki, 2018.
- [HKLS15] H. Hoos, R. Kaminski, M. Lindauer, and T. Schaub. aspeed: Solver scheduling via answer set programming. *Theory and Practice of Logic Programming*, 15:117–142, 2015.
- [HLB⁺17] F. Hutter, M. Lindauer, A. Balint, S. Bayless, H. Hoos, and K. Leyton-Brown. The configurable SAT solver challenge (CSSC). *Artificial Intelligence Journal (AIJ)*, 243:1–25, February 2017.
- [HLS14] H. Hoos, M. Lindauer, and T. Schaub. claspfolio 2: Advances in algorithm selection for answer set programming. *Theory and Practice of Logic Programming*, 14:569–585, 2014.
- [Hoo12] H. Hoos. Programming by optimization. *Communications of the ACM*, 55(2):70–80, 2012.
- [HTH02] F. Hutter, D. A. D. Tompkins, and H. H. Hoos. Scaling and probabilistic smoothing: efficient dynamic local search for SAT. In *Proceedings of the 8th International conference on Principles and Practice of Constraint Programming (CP'02)*, volume 2470 of *Lecture Notes in Computer Science*, pages 233–248. Springer Verlag, 2002.
- [HXHLB14] F. Hutter, L. Xu, H. Hoos, and K. Leyton-Brown. Algorithm runtime prediction: Methods and evaluation. *Artificial Intelligence*, 206:79–111, 2014.
- [JBRS12] M. Järvisalo, D. Le Berre, O. Roussel, and L. Simon. The international SAT solver competitions. *AI Magazine*, 33(1):89–94, 2012.
- [JSW98] D. R. Jones, M. Schonlau, and W. J. Welch. Efficient global optimization of expensive black box functions. *Journal of Global Optimization*, 13:455–492, 1998.

- [KFM⁺18] L. Kotthoff, A. Fréchet, T.P. Michalak, T. Rahwan, H. Hoos, and K. Leyton-Brown. Quantifying algorithmic improvements over time. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI'18)*, pages 5165–5171, 2018.
- [KLBL17] R. Kleinberg, K. Leyton-Brown, and B. Lucier. Efficiency through procrastination: Approximately optimal algorithm configuration with runtime guarantees. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*, 2017.
- [KLBLG19] R. Kleinberg, K. Leyton-Brown, B. Lucier, and D. Graham. Procrastinating with confidence: Near-optimal, anytime, adaptive algorithm configuration. *arXiv:1902.05454 [cs.AI]*, 2019.
- [KMS⁺11] S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann. Algorithm selection and scheduling. In *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP'11)*, volume 6876 of *Lecture Notes in Computer Science*, pages 454–469. Springer, 2011.
- [KMST10] S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney. ISAC - instance-specific algorithm configuration. In *Proceedings of the Nineteenth European Conference on Artificial Intelligence (ECAI'10)*, pages 751–756. IOS Press, 2010.
- [Knu15] D.E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Addison-Wesley, Reading, MA, USA, 2015.
- [KXHLB09] A. KhudaBukhsh, L. Xu, H.H. Hoos, and K. Leyton-Brown. SATenstein: Automatically building local search SAT solvers from components. In *Proceedings of the 22th International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 517–524, 2009.
- [KXHLB16] A. KhudaBukhsh, L. Xu, H.H. Hoos, and K. Leyton-Brown. SATenstein: Automatically building local search SAT solvers from components. *Artificial Intelligence Journal (AIJ)*, 232:20–42, March 2016.
- [LBNA⁺03] K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham. Boosting as a metaphor for algorithm design. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP'03)*, volume 2470 of *Lecture Notes in Computer Science*, pages 899–903. Springer Verlag, 2003.
- [LBNS09] K. Leyton-Brown, E. Nudelman, and Y. Shoham. Empirical hardness models: Methodology and a case study on combinatorial auctions. *Journal of the ACM*, 56(4):1–52, 2009.
- [LH18] C. Luo and H.H. Hoos. Sparkle SAT Challenge 2018, 2018. <http://ada.liacs.nl/events/sparkle-sat-18>. Last accessed on 15 June 2019.
- [LHH15] M. Lindauer, H. Hoos, and F. Hutter. From sequential algorithm selection to parallel portfolio selection. In *Proceedings of the Ninth International Conference on Learning and Intelligent Optimization (LION'15)*, volume 8994 of *Lecture Notes in Computer Science*, pages 1–16. Springer Verlag, 2015.
- [LHHS15] M. Lindauer, H. Hoos, F. Hutter, and T. Schaub. Autofolio: An automatically configured algorithm selector. *Journal of Artificial*

Intelligence Research, 53:745–778, August 2015.

- [LHLBS17] M. Lindauer, H. Hoos, K. Leyton-Brown, and T. Schaub. Automatic construction of parallel portfolios via algorithm configuration. *Artificial Intelligence Journal (AIJ)*, 244:272–290, March 2017.
- [LIDLSB11] M. López-Ibáñez, J. Dubois-Lacoste, T. Stützle, and M. Birattari. The irace package, iterated race for automatic algorithm configuration. Technical report, IRIDIA, Université Libre de Bruxelles, Belgium, 2011.
- [LL00] M.G. Lagoudakis and M.L. Littman. Algorithm selection using reinforcement learning. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML'00)*, pages 511–518, 2000.
- [LL01] M.G. Lagoudakis and M.L. Littman. Learning to select branching rules in the DPLL procedure for satisfiability. In *Electronic Notes in Discrete Mathematics*, volume 9, pages 344–359, 2001.
- [LMS02] H. R. Lourenço, O. Martin, and T. Stützle. Iterated local search. In *Handbook of Metaheuristics*, pages 321–353. 2002.
- [Man13] N. Manthey. The SAT solver RISS3G at SC 2013. In *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of *Department of Computer Science Series of Publications B*, pages 72–73. University of Helsinki, 2013.
- [ML16] N. Manthey and M. Lindauer. Spybug: Automated bug detection in the configuration space of SAT solvers. In *Proceedings of the International Conference on Satisfiability Solving (SAT'16)*, pages 554–561, 2016.
- [MS14] N. Manthey and P. Steinke. Too many rooks. In *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions*, volume B-2014-2 of *Department of Computer Science Series of Publications B*, pages 97–98. University of Helsinki, 2014.
- [MSSS13] Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann. Algorithm portfolios based on cost-sensitive hierarchical clustering. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence (IJCAI'13)*, pages 608–614. AAAI Press, 2013.
- [NFLB17] N. Newman, A. Fréchette, and K. Leyton-Brown. Deep optimization for spectrum repacking. *Communications of the ACM*, 61(1):97–104, 2017.
- [NLBA⁺03] E. Nudelman, K. Leyton-Brown, G. Andrew, C. Gomes, J. McFadden, B. Selman, and Y. Shoham. Satzilla 0.9. Solver description, International SAT Competition, 2003.
- [NLBH⁺04] E. Nudelman, K. Leyton-Brown, H. H. Hoos, A. Devkar, and Y. Shoham. Understanding random SAT: beyond the clauses-to-variables ratio. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP'04)*, volume 3258 of *Lecture Notes in Computer Science*, pages 438–452. Springer Verlag, 2004.
- [NLBMS17] N. Newman, K. Leyton-Brown, P. Milgrom, and I. Segal. Assessing economic outcomes in simulated reverse clock auctions for radio spectrum. *arXiv:1706.04324 [cs.GT]*, 2017.

- [PCLHS17] L. Pérez Cáceres, M. López-Ibáñez, H. Hoos, and T. Stützle. An experimental study of adaptive capping in irace. In *Proceedings of the 11th International Conference on Learning and Intelligent Optimization (LION 11)*, volume 10556 of *Lecture Notes in Computer Science*, pages 235–250. Springer, 2017.
- [Ric76] J. R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.
- [Rou11] O. Roussel. Description of ppfolio. Solver description, SAT competition 2011, 2011.
- [SBH05] L. Simon, D. Le Berre, and E. Hirsch. The SAT2002 competition report. *Annals of Mathematics and Artificial Intelligence*, 43:307–342, 2005.
- [SH79] J. Schmee and G. J. Hahn. A simple method for regression analysis with censored data. *Technometrics*, 21(4):417–432, 1979.
- [SLA12] J. Snoek, H. Larochelle, and R. P. Adams. Practical Bayesian optimization of machine learning algorithms. In *Proceedings of the 26th International Conference on Advances in Neural Information Processing Systems (NIPS’12)*, pages 2960–2968, 2012.
- [SLS12] B. Silverthorn, Y. Lierler, and M. Schneider. Surviving solver sensitivity: An ASP practitioner’s guide. In *Technical Communications of the Twenty-eighth International Conference on Logic Programming (ICLP’12)*, volume 17, pages 164–175. Leibniz International Proceedings in Informatics (LIPIcs), 2012.
- [SSHH15] J. Seipp, S. Sievers, M. Helmert, and F. Hutter. Automatic configuration of sequential planning portfolios. In *Proceedings of the Twenty-ninth National Conference on Artificial Intelligence (AAAI’15)*, pages 3364–3370. AAAI Press, 2015.
- [THHLB13] C. Thornton, F. Hutter, H. Hoos, and K. Leyton-Brown. AutoWEKA: combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD’13)*, pages 847–855. ACM press, New York, NY, USA, 2013.
- [VFG+13] M. Vallati, C. Fawcett, A. Gerevini, H. Hoos, and A. Saetti. Automatic generation of efficient domain-optimized planners from generic parametrized planners. In *Proceedings of the Sixth Annual Symposium on Combinatorial Search (SOCS’14)*. AAAI Press, 2013.
- [WGS18a] G. Weisz, A. Gyögy, and C. Szepesvári. CAPSANDRUNS: An improved method for approximately optimal algorithm configuration. *ICML 2018 AutoML Workshop*, 2018.
- [WGS18b] G. Weisz, A. Gyögy, and C. Szepesvári. LEAPSANDBOUNDS: A method for approximately optimal algorithm configuration. In *Proceedings of the 35th International Conference on Machine Learning (ICML’18)*, pages 5254–5262, 2018.
- [XHHLB07] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown. Satzilla-07: The design and analysis of an algorithm portfolio for SAT. In *Proceedings of the Thirteenth International Conference on Principles and Practice of Constraint Programming (CP’07)*, volume 4741 of *Lec-*

ture Notes in Computer Science, pages 712–727. Springer Verlag, 2007.

- [XHHLB08] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, June 2008.
- [XHHLB09] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown. SATzilla2009: an automatic algorithm portfolio for sat. Solver description, SAT competition 2009, 2009.
- [XHHLB12] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Evaluating component solver contributions in portfolio-based algorithm selectors. In *Proceedings of the Fifteenth International Conference on Theory and Applications of Satisfiability Testing (SAT'12)*, volume 7317 of *Lecture Notes in Computer Science*, pages 228–241. Springer Verlag, 2012.
- [XHLB07] L. Xu, H. H. Hoos, and K. Leyton-Brown. Hierarchical hardness models for SAT. In *Proceedings of the Thirteenth International Conference on Principles and Practice of Constraint Programming (CP'07)*, volume 4741 of *Lecture Notes in Computer Science*, pages 696–711. Springer Verlag, 2007.
- [XHLB10] L. Xu, H. H. Hoos, and K. Leyton-Brown. Hydra: Automatically configuring algorithms for portfolio-based selection. In *Proceedings of the Twenty-fourth National Conference on Artificial Intelligence (AAAI'10)*, pages 210–216. AAAI Press, 2010.

This page intentionally left blank

Chapter 13

Symmetry and Satisfiability

Karem A. Sakallah

Symmetry is at once a familiar concept (we recognize it when we see it!) and a profoundly deep mathematical subject. At its most basic, a symmetry is some transformation of an object that leaves the object (or some aspect of the object) unchanged. For example, Fig. 13.1 shows that a square can be transformed in eight different ways that leave it looking exactly the same: the identity “do-nothing” transformation, 3 rotations, and 4 mirror images (or reflections). In the context of decision problems, the presence of symmetries in a problem’s search space can frustrate the hunt for a solution by forcing a search algorithm to fruitlessly explore symmetric subspaces that do not contain solutions. Recognizing that such symmetries exist, we can direct a search algorithm to look for solutions only in non-symmetric parts of the search space. In many cases, this can lead to significant pruning of the search space and yield solutions to problems which are otherwise intractable.

In this chapter we will be concerned with the symmetries of Boolean functions, particularly the symmetries of their conjunctive normal form (CNF) representations. Our goal is to understand what those symmetries are, how to model them using the mathematical language of group theory, how to derive them from a CNF formula, and how to utilize them to speed up CNF SAT solvers.

In a surprising twist, the quest for exploiting symmetries to improve the performance of SAT solvers led to the development of a new highly-scalable symmetry detection algorithm inspired by the backtrack search architecture of SAT solvers! The chapter provides an informal description of this algorithm and concludes with a brief review of symmetry applications in a number of fields.

13.1. Motivating Example

Consider the Boolean function $f(a, b, c)$ specified in Table 13.1. It can be expressed algebraically in sum-of-minterms form as

$$f(a, b, c) = a'bc + ab'c \tag{13.1}$$

It is not too difficult to observe that this function remains invariant under a swap of inputs a and b :

$$f(b, a, c) = b'ac + ba'c = ab'c + a'bc = a'bc + ab'c \tag{13.2}$$

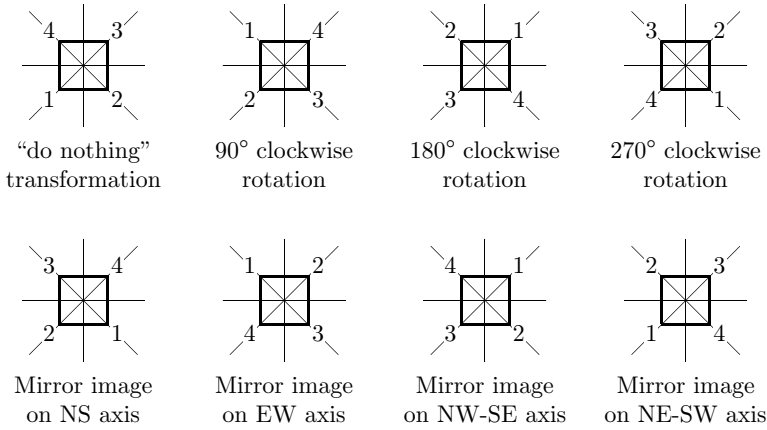


Figure 13.1. Symmetries of the square

It is also rather obvious that this invariance follows from the commutativity of the Boolean AND and OR operators. Viewing this function as an “object” with three “features” labeled a , b , and c , we may consider this swap as a transformation that leaves the essential nature of the object (its “function”) unchanged, hence a symmetry of the object. Allowing for possible inversion, in addition to swaps, of the inputs we can readily check that the following two transformations are also symmetries of this function:

$$f(b', a', c) = b''a'c + b'a''c = a'bc + ab'c \quad (13.3)$$

$$f(a', b', c) = a''b'c + a'b''c = ab'c + a'bc = a'bc + ab'c \quad (13.4)$$

Besides commutativity, invariance under these two transformations relies on the involution of double complementation, i.e., the fact that $x'' = x$. Note that the transformation in (13.3) can be described as a swap between a and b' (equivalently between b and a'). On the other hand, the transformation in (13.4) is a *simultaneous swap* between a and a' , and between b and b' .

To facilitate later discussion, let us label these swaps as follows:

Table 13.1. Truth table for example Boolean function (13.1)

a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

Table 13.2. Symmetry composition table for example function (13.1). The entry in row r and column c represents the result of applying symmetry c followed by symmetry r .

$r * c$	ε	α	β	γ
ε	ε	α	β	γ
α	α	ε	γ	β
β	β	γ	ε	α
γ	γ	β	α	ε

Label	Swap
ε	No swap
α	Swap a with b
β	Swap a with b'
γ	Swap a with a' and b with b'

A natural question now arises: are these the only possible transformations that leave this function unchanged? At this point we cannot answer the question definitively. However, we can try to verify that combining the above transformations will not yield new transformations. Specifically, if we let $\sigma * \tau$ denote the application of τ followed by σ we obtain the *symmetry composition table* in Table 13.2¹. Thus the set of transformations $\{\varepsilon, \alpha, \beta, \gamma\}$ is *closed* under the composition operator $*$.

To visualize this function's symmetries, consider its Karnaugh map (K-map) in Table 13.3. The map clearly shows that the function's solution space (the 8 possible truth assignments to the function's three propositional variables) consists of two symmetric subspaces. Specifically, the distribution of the function's 1s and 0s in the subspace where $a = 0$ is identical to their distribution in the subspace where $a = 1$. Stated another way, the function's 8 possible truth assignments can be collapsed to just these 4 equivalence classes: $\{0, 6\}$, $\{1, 7\}$, $\{2, 4\}$, and $\{3, 5\}$. The assignments in each such class are equivalent in the sense that they yield the same function value (either 0 or 1). Thus, to determine the satisfiability of the function, it is sufficient to consider only one such assignment from each equivalence class. This can be accomplished by conjoining the function with a constraint that filters out all but the chosen representative assignment from each class. A convenient filter for this example is either the constraint (a), which restricts the search for a satisfying assignment to the bottom 4 cells of the map, or the constraint (a'), which restricts the search to the top 4 cells of the map². These filters are referred to as *symmetry-breaking predicates* (SBPs for short) because they serve to block a search algorithm from exploring symmetric assignments in the solution space.

The rest of this chapter is divided into ten sections. In Sec. 13.2 we cover some preliminaries related to Boolean algebra and set partitions. Sec. 13.3 introduces the basics of group theory necessary for representing and manipulating the symmetries of Boolean functions. This is further articulated in Sec. 13.4 for the

¹Some authors define composition using the opposite order: $\sigma * \tau$ means "apply σ followed by τ ." This does not matter as long as the operation order is defined consistently.

²Other, less convenient, filters include $(a'c' + ac)$ which selects assignments $\{0, 2, 5, 7\}$ and excludes assignments $\{1, 3, 4, 6\}$.

Table 13.3. K-map for example function (13.1). The small numbers in the map cells indicate the decimal index of the corresponding minterm (variable assignment) assuming that a is the most significant bit and c is the least significant bit.

		c		
		0	1	
	00	0 ⁰	0 ¹	
	01	0 ²	1 ³	b
	11	0 ⁶	0 ⁷	
a	10	0 ⁴	1 ⁵	

CNF representation of such functions. In Sec. 13.5 we briefly describe the colored graph automorphism problem and the basic algorithm for solving it. This sets the stage, in Sec. 13.6, for the discussion on detecting the symmetries of CNF formulas by reduction to graph automorphism. The construction of symmetry breaking predicates is covered in Sec. 13.7. Sec. 13.8 describes the SAT-inspired **saucy** graph automorphism algorithm. Sec. 13.9 summarizes the whole symmetry detection and breaking flow, as well as offering some ideas on an alternative approach that requires further exploration. The chapter concludes in Sec. 13.10 with a historical review of relevant literature.

13.2. Preliminaries

Since we will be mostly concerned with symmetries in the context of Boolean satisfiability, it is useful to establish in this section the vocabulary of Boolean algebra to which we will make frequent reference. Specifically, we will be exclusively dealing with the 2-valued Boolean algebra, also known as propositional logic, which we define as follows:

Definition 13.2.1. (2-Valued Boolean Algebra) The 2-valued Boolean algebra is the algebraic structure $\langle \{0, 1\}, \cdot, +, ' \rangle$ where $\{0, 1\}$ is the set of *truth values*, and where the binary operators \cdot and $+$, and the unary operator $'$ are defined by the following rules:

- AND (logical conjunction): $0 \cdot 0 = 0 \cdot 1 = 1 \cdot 0 = 0, 1 \cdot 1 = 1$
- OR (logical disjunction): $0 + 0 = 0, 0 + 1 = 1 + 0 = 1 + 1 = 1$
- NOT (logical negation): $0' = 1, 1' = 0$

We will, on occasion, use other symbols to indicate these operators such as \wedge for AND, \vee for OR, and \neg for NOT. In addition, we will follow custom by indicating the AND operator with a simple juxtaposition of its arguments.

A Boolean *variable* is a variable whose domain is the set $\{0, 1\}$. Although not formally synonymous, we may at times refer to such variables as *binary* or as *propositional*. A Boolean *literal* is either a Boolean variable or the negation of a Boolean variable. The domain of a vector of n Boolean variables is the set

$\{0, 1\}^n$ whose elements are the 2^n ordered n -tuples, or combinations, of 0 and 1. In other words,

$$\{0, 1\}^n = \left\{ \underbrace{(0, 0, \dots, 0, 0)}_{n \text{ bits}}, \underbrace{(0, 0, \dots, 0, 1)}_{n \text{ bits}}, \dots, \underbrace{(1, 1, \dots, 1, 1)}_{n \text{ bits}} \right\}$$

Each of these combinations can also be interpreted naturally as an unsigned n -bit binary integer. Expressing each of these integers in decimal notation, we obtain the following equivalent representation of this domain:

$$\{0, 1\}^n = \{0, 1, 2, \dots, 2^n - 1\}$$

We will refer to elements of this set as *truth assignments* or “points” in the 2^n -dimensional Boolean space and identify each by its decimal “index”.

Individual Boolean variables will be denoted by lower-case letters, possibly with a numeric subscript, e.g., a, b, x_3 , etc. When speaking about the literals of a particular variable we will place a dot over the variable symbol. Thus, the literal \dot{a} stands for either a or a' . Vectors of Boolean variables will be denoted by upper case letters and, as above, interpreted as unsigned binary integers. Thus, $X = (x_{n-1}, x_{n-2}, \dots, x_0)$ is a vector of n Boolean variables representing an unsigned n -bit integer whose most significant bit is x_{n-1} and whose least significant bit is x_0 . An assignment $X = i$ to an n -variable Boolean vector where $0 \leq i \leq 2^n - 1$ is short-hand for assigning to the elements of X the appropriate bits of the binary integer corresponding to i . Thus, for $n = 3$, $X = 5$ means $x_2 = 1$, $x_1 = 0$ and $x_0 = 1$.

An n -variable Boolean function is defined by the map $\{0, 1\}^n \rightarrow \{0, 1\}$, i.e., each of the 2^n points in its domain is mapped to one of the two binary values. There are two classes of elementary Boolean functions: minterms and maxterms.

Definition 13.2.2. (Minterms and Maxterms) The i th n -variable *minterm function*, denoted by $m_i(X)$, maps point i in the 2^n -dimensional Boolean space to 1, and maps all other points to 0. The i th n -variable *maxterm function*, denoted by $M_i(X)$, maps point i in the 2^n -dimensional Boolean space to 0, and maps all other points to 1. Algebraically, an n -variable minterm function is an AND expression of n distinct literals, whereas an n -variable maxterm function is an OR expression of n distinct literals.

AND and OR expressions (not necessarily consisting of all n literals) are commonly referred to as *product* and *sum terms*, respectively.

Minterm and maxterm functions serve as building blocks for representing any n -variable Boolean function. For example, the function specified in Table 13.1 has the following *sum-of-minterm* and *product-of-maxterm* representations:

$$\begin{aligned} f(a, b, c) &= m_3 + m_5 = a'bc + ab'c \\ &= M_0 \cdot M_1 \cdot M_2 \cdot M_4 \cdot M_6 \cdot M_7 \\ &= (a + b + c)(a + b + c')(a + b' + c)(a' + b + c)(a' + b' + c)(a' + b' + c') \end{aligned}$$

Examination of the sum-of-minterms expression shows that it evaluates to 1 for combinations 3 and 5 on variables (a, b, c) and evaluates to 0 for all other combinations. Informally, we say that function f consists of minterms 3 and 5. Alternatively, we can say that f consists of maxterms 0, 1, 2, 4, 6, and 7. A commonly-used short-hand for representing Boolean functions this way is the following:

$$f(a, b, c) = \sum_{(a,b,c)} (3, 5) = \prod_{(a,b,c)} (0, 1, 2, 4, 6, 7)$$

Definition 13.2.3. (Implication) Given two Boolean functions $f(X)$ and $g(X)$, we say that $g(X)$ *implies* $f(X)$, written $g(X) \rightarrow f(X)$, if all of g 's minterms are also minterms of f .

Definition 13.2.4. (Implicants and Implicates) A product term $p(X)$ is an *implicant* of a function $f(X)$ if $p(X) \rightarrow f(X)$. A sum term $s(X)$ is an *implicate* of a function $f(X)$ if $f(X) \rightarrow s(X)$.

A function can have many implicants and implicates. In particular, a function's implicants include its minterms and its implicates include its maxterms.

Definition 13.2.5. (Disjunctive and Conjunctive Normal Forms) A *disjunctive normal form* (DNF) expression of a Boolean function is an OR of implicants. It is also known as a sum-of-products (SOP). A *conjunctive normal form* (CNF) expression of a Boolean function is an AND of implicates. It is also known as a product-of-sums (POS). It is customary to refer to the implicates in a CNF representation as *clauses*.

Definition 13.2.6. (Minimal DNF and CNF Expressions) The *cost* of a DNF or CNF expression $\varphi(X)$ is the tuple $(\text{terms}(\varphi), \text{literals}(\varphi))$ where $\text{terms}(\varphi)$ and $\text{literals}(\varphi)$ are, respectively, the number of terms (implicants or implicates) and total number of literals in the expression. We say that expression φ is *cheaper* than expression ϑ if $\text{terms}(\varphi) \leq \text{terms}(\vartheta)$ and $\text{literals}(\varphi) < \text{literals}(\vartheta)$. A DNF expression $\varphi(X)$ is minimal for function $f(X)$ if there are no other DNF expressions for $f(X)$ that are cheaper than $\varphi(X)$. Similarly, A CNF expression $\varphi(X)$ is minimal for function $f(X)$ if there are no other CNF expressions for $f(X)$ that are cheaper than $\varphi(X)$. Note that this definition of cost induces a partial order on the set of equivalent DNF (resp. CNF) expressions of a Boolean function.

Definition 13.2.7. (Prime Implicants and Prime Implicates) A *prime implicant* $p(X)$ of a function $f(X)$ is an implicant that is not contained in any other implicant of $f(X)$. In other words, $p(X)$ is a prime implicant of $f(X)$ if $p(X) \rightarrow f(X)$ and $p(X) \not\rightarrow q(X)$ where $q(X)$ is an implicant of $f(X)$. A *prime implicate* $s(X)$ of a function $f(X)$ is an implicate that does not contain any other implicate of $f(X)$. In other words, $s(X)$ is a prime implicate of $f(X)$ if $f(X) \rightarrow s(X)$ and $t(X) \not\rightarrow s(X)$ where $t(X)$ is an implicate of $f(X)$.

Theorem 13.2.1. (Prime Implicant/Prime Implicate Theorem [Qui52, Qui55, Qui59]) Any minimal DNF expression of a function $f(X)$ can only consist of prime implicants. Similarly, any minimal CNF expression can only consist of prime implicates. Minimal expressions are not necessarily unique.

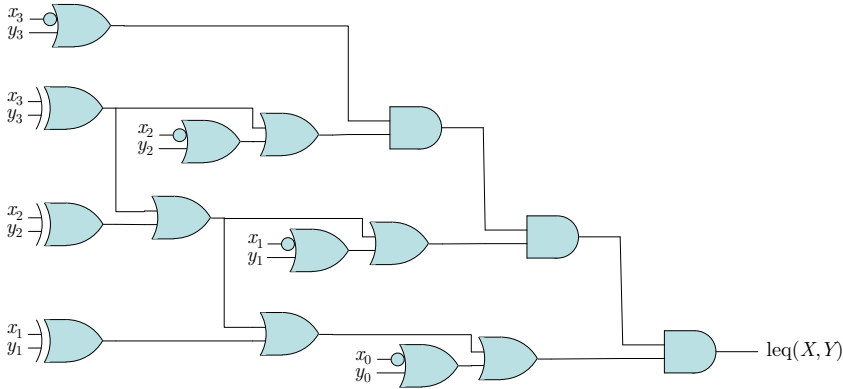


Figure 13.2. A 4-bit comparator circuit

Definition 13.2.8. (leq predicate) The *less-than-or-equal* predicate in the 2^n -dimensional Boolean space is defined by the map

$$\begin{aligned} \text{leq}(X, Y) &= X \leq Y \\ &= \bigwedge_{i=0}^{n-1} \left[\left[\bigwedge_{j=i+1}^{n-1} (x_j = y_j) \right] \rightarrow (x_i \leq y_i) \right] \end{aligned} \quad (13.5)$$

This is a common arithmetic function that is usually implemented in a multi-level circuit similar to that shown in Fig. 13.2. In comparing the circuit to equation (13.5), note that $(x \rightarrow y) = (x \leq y) = (x' + y)$ and $(x = y)' = (x \oplus y)$, where \oplus denotes *exclusive OR*. Note also that the leq predicate imposes a total ordering on the 2^n truth assignments that conforms to their interpretation as unsigned integers. As will become clear later, the leq predicate provides one mechanism for breaking symmetries.

We close with a brief mention of set partitions since they arise in the context of describing and detecting the symmetries of a CNF formula.

Definition 13.2.9. (Partition) A partition π of a set X is a set of non-empty subsets of X , $\{X_i \mid X_i \subseteq X, i = 1, \dots, k\}$, such that:

$$\begin{aligned} X_i \cap X_j &= \emptyset, \text{ for } i \neq j \\ \bigcup_{1 \leq i \leq k} X_i &= X \end{aligned}$$

The elements of π are variously referred to as the *blocks*, *classes* or *cells* of the partition. A partition is *discrete* if all of its cells are singleton sets, and *unit* if it has only one cell (the whole set).

Two elements $x, y \in X$ are equivalent under partition π , written as $x \sim y$, if $x, y \in X_i$ for some $i \in [1, k]$.

A partition is usually written as a set of sets. For example, $\{\{1, 2\}, \{4\}, \{3, 5\}\}$ is a partition of $\{1, \dots, 5\}$. A slightly more compact notation for this partition uses vertical bars to separate the cells: $\{1, 2|4|3, 5\}$. When the order of the cells in a partition is important, curly brackets are replaced by square brackets. For example, $[1, 2|4|3, 5]$ is an *ordered partition* which is different from $[4|3, 5|1, 2]$.

Definition 13.2.10. (Lattice of Partitions [Can01]) Given two partitions π and π' of a set X , we say that π' is *finer* than π , written as $\pi' \leq \pi$, if every cell of π' is contained in some cell of π . The relation “is finer than” defines a *lattice* on the set of partitions whose top element is the unit partition and whose bottom element is the discrete partition. Given two partitions π_1 and π_2 , their *intersection*, $\pi_1 \cap \pi_2$, is the *largest* partition that refines both. Dually, the *union* of π_1 and π_2 , $\pi_1 \cup \pi_2$, is the *smallest* partition which is refined by both.

Example 13.2.1. Given $X = \{1, 2, \dots, 10\}$ and the two partitions

$$\begin{aligned}\pi_1 &= \{1, 2, 4|3|5|6, 7, 8|9, 10\} \\ \pi_2 &= \{1, 3, 4|2|5, 6, 7, 8|9|10\}\end{aligned}$$

their intersection and union are

$$\begin{aligned}\pi_1 \cap \pi_2 &= \{1, 4|2|3|5|6, 7, 8|9|10\} \\ \pi_1 \cup \pi_2 &= \{1, 2, 3, 4|5, 6, 7, 8|9, 10\}\end{aligned}$$

13.3. Group Theory Basics

The study of symmetries in Boolean functions and their various representations is greatly facilitated by understanding some basic notions from the vast field of group theory. In this section we cover only those aspects of group theory that are necessary to follow the development, in subsequent sections, of the symmetry detection and breaking approaches for CNF representations of Boolean functions. More comprehensive treatments of group theory are available in standard textbooks; a particularly accessible source is [Fra00] which we liberally borrow from in what follows.

13.3.1. Groups

We begin with the definition of a group as an abstract algebraic structure with certain properties.

Definition 13.3.1. (Group) A group is a structure $\langle G, * \rangle$, where G is a (non-empty) set that is closed under a binary operation $*$, such that the following axioms are satisfied:

- The operation is *associative*: for all $x, y, z \in G$, $(x * y) * z = x * (y * z)$
- There is an *identity* element $e \in G$ such that: for all $x \in G$, $e * x = x * e = x$
- Every element $x \in G$ has an inverse $x^{-1} \in G$ such that: $x * x^{-1} = x^{-1} * x = e$

We will typically refer to the group G , rather than to the structure $\langle G, * \rangle$, with the understanding that there is an associated binary operation on the set G . Furthermore, it is customary to write xy instead of $x * y$ and x^2 instead of $x * x$ much like we do with the multiplication operation on numbers.

This definition leads to a number of interesting properties that are easy to prove, including:

- **Uniqueness of the identity:** there is only one element $e \in G$ such that $e * x = x * e = x$ for all $x \in G$, and
- **Uniqueness of the inverse:** for each $x \in G$, there is only one element $x^{-1} \in G$ such that $x * x^{-1} = x^{-1} * x = e$.

The definition also does not restrict a group to be finite. However, for our purposes it is sufficient to focus on finite groups.

Definition 13.3.2. (Group Order) If G is a finite group, its *order* $|G|$ is the number of elements in (i.e., the cardinality of) the set G .

Under these two definitions, the set of transformations introduced in Sec. 13.1 is easily seen to be a group of order 4 with group operation $*$ as defined in Table 13.2. The identity element is the “do nothing” transformation ε and each of the four elements in the group is its own inverse. The associativity of the operation can be verified by enumerating all possible triples from $\{\varepsilon, \alpha, \beta, \gamma\}^3$. This group is known as the Klein 4-group V and is one of only two groups of order 4 [Fra00, p. 67].

Definition 13.3.3. (Group Isomorphism) Let $\langle G, * \rangle$ and $\langle G', *' \rangle$ be two groups. An *isomorphism* of G with G' is a one-to-one function ϕ mapping G onto G' such that:

$$\phi(x * y) = \phi(x) *' \phi(y) \text{ for all } x, y \in G$$

If such a map exists, then G and G' are *isomorphic groups* which we denote by $G \simeq G'$.

To illustrate group isomorphism, consider the following set of all possible negations (complementations) of two binary literals \dot{a}, \dot{b} :

Label	Negation
$\dot{a}\dot{b}$	Don't negate either literal
$\dot{a}'\dot{b}$	Only negate \dot{a}
$\dot{a}\dot{b}'$	Only negate \dot{b}
$\dot{a}'\dot{b}'$	Negate both \dot{a} and \dot{b}

The labels in the above table should be viewed as mnemonics that indicate the particular transformation on the two literals: a prime on a literal means the literal should be complemented by the transformation, and the absence of a prime means the literal should be left unchanged. As such, these transformations can

³Such enumeration is impractical or even infeasible for large groups. In general, proving that the group operation is associative is typically done by showing that the group is isomorphic (see Def. 13.3.3) to another group whose operation is *known* to be associative.

Table 13.4. Composition table for \mathcal{N}_2

$*'$	$\dot{a}\dot{b}$	$\dot{a}'\dot{b}$	$\dot{a}\dot{b}'$	$\dot{a}'\dot{b}'$
$\dot{a}\dot{b}$	$\dot{a}\dot{b}$	$\dot{a}'\dot{b}$	$\dot{a}\dot{b}'$	$\dot{a}'\dot{b}'$
$\dot{a}'\dot{b}$	$\dot{a}'\dot{b}$	$\dot{a}\dot{b}$	$\dot{a}'\dot{b}'$	$\dot{a}\dot{b}'$
$\dot{a}\dot{b}'$	$\dot{a}\dot{b}'$	$\dot{a}'\dot{b}'$	$\dot{a}\dot{b}$	$\dot{a}'\dot{b}$
$\dot{a}'\dot{b}'$	$\dot{a}'\dot{b}'$	$\dot{a}\dot{b}$	$\dot{a}'\dot{b}$	$\dot{a}\dot{b}'$

be combined. For example, negating both literals followed by negating only \dot{a} yields the same result as negating just \dot{b} . Denoting this operation by $*'$ we obtain the composition table in Table 13.4. Clearly, as defined, the set $\{\dot{a}\dot{b}, \dot{a}'\dot{b}, \dot{a}\dot{b}', \dot{a}'\dot{b}'\}$ along with the operation $*'$ is a group. Let's denote this group by \mathcal{N}_2 (for the *group of negations on two binary literals*). Close examination of Table 13.2 and Table 13.4 shows that, other than the particular labels used, they have identical structures. Specifically, we have the following map between the Klein 4-group V and the group of negations \mathcal{N}_2 :

$$\begin{array}{lcl}
 x \in V & \leftrightarrow & \phi(x) \in \mathcal{N}_2 \\
 \varepsilon & \leftrightarrow & \dot{a}\dot{b} \\
 \alpha & \leftrightarrow & \dot{a}'\dot{b} \\
 \beta & \leftrightarrow & \dot{a}\dot{b}' \\
 \gamma & \leftrightarrow & \dot{a}'\dot{b}'
 \end{array}$$

Thus V and \mathcal{N}_2 are isomorphic groups. In what follows we will use \mathcal{N}_n to denote the group of negations on n literals. This group has order 2^n .

13.3.2. Subgroups

Subgroups provide a way to understand the structure of groups.

Definition 13.3.4. (Subgroup) A non-empty subset H of a group G that is closed under the binary operation of G is itself a group and is referred to as a *subgroup* of G . We indicate that H is a subgroup of G by writing $H \leq G$. Additionally, $H < G$ shall mean that $H \leq G$ but $H \neq G$.

Definition 13.3.5. (Proper and Trivial Subgroups) If G is a group, then the subgroup consisting of G itself is the *improper subgroup* of G . All other subgroups are *proper subgroups*. The group $\{e\}$ is the *trivial subgroup* of G . All other subgroups are *non-trivial*.

We illustrate the notion of subgroups for the Klein 4-group introduced earlier. Specifically, this group has four non-trivial subgroups: $\{\varepsilon, \alpha\}$, $\{\varepsilon, \beta\}$, $\{\varepsilon, \gamma\}$, as well as the whole group itself. These groups are related according to the lattice structure shown in Fig. 13.3.

A subgroup H of a group G induces a partition of G whose cells are referred to as the *cosets* of H :

Definition 13.3.6. (Cosets) Let H be a subgroup of a group G . The *left coset* of H containing $x \in G$ is the subset $xH = \{xy | y \in H\}$ and the *right coset* of H containing $x \in G$ is the subset $Hx = \{yx | y \in H\}$.

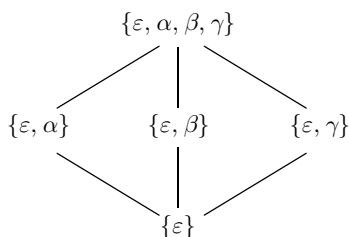


Figure 13.3. Lattice diagram for the Klein 4-group V showing its subgroups.

For example, the left cosets of $H = \{\varepsilon, \alpha\}$ in the Klein-4 group $\{\varepsilon, \alpha, \beta, \gamma\}$ yield the partition $\{\{\varepsilon, \alpha\}, \{\beta, \gamma\}\}$.

It is easily shown that the number of elements in each coset of a subgroup H of a group G is the same as the number of elements of H [Fra00, p. 124]. This immediately leads to the following fundamental theorem of group theory:

Theorem 13.3.1. (Theorem of Lagrange) Let H be a subgroup of a finite group G . Then $|H|$ is a divisor of $|G|$.

13.3.3. Group Generators

Theorem 13.3.2. (Cyclic Subgroup) Let G be a group and let $x \in G$. Then

$$H = \{x^n \mid n \in \mathbb{Z}\}$$

i.e., the set of all (not necessarily distinct) powers of x , is a subgroup of G and is the smallest subgroup of G that contains x . This is referred to as the *cyclic subgroup of G generated by x* , and is denoted by $\langle x \rangle$.

Definition 13.3.7. (Generator; Cyclic Group) An element x of a group G *generates* G and is a *generator* for G if $\langle x \rangle = G$. A group G is *cyclic* if there is some element $x \in G$ that generates G .

Definition 13.3.8. (Group Generators) Let G be a group and let $H \subset G$ be a subset of the group elements. The smallest subgroup of G containing H is the *subgroup generated by H* . If this subgroup is all of G , then H *generates* G and the elements of H are *generators* of G . A generator is *redundant* if it can be expressed in terms of other generators. A set of generators for a group G is *irredundant* if it does not contain redundant generators.

An important property of irredundant generator sets of a finite group is that they provide an extremely compact representation of the group. This follows directly from the Theorem of Lagrange and the definition of irredundant generator sets:

Theorem 13.3.3. Any irredundant set of generators for a finite group G , such that $|G| > 1$, contains at most $\log_2 |G|$ elements.

Proof. Let $\{x_i \in G \mid 1 \leq i \leq n\}$ be a set of n irredundant generators for group G and consider the sequence of subgroups G_1, G_2, \dots, G_n such that group G_i is

generated by $\{x_j \in G \mid 1 \leq j \leq i\}$. Clearly, $G_1 < G_2 < \dots < G_n$ because of our assumption that the generators are irredundant. By the theorem of Lagrange, $|G_n| \geq 2|G_{n-1}| \geq \dots \geq 2|G_2| \geq 2|G_1|$, i.e., $|G_n| \geq 2^{n-1}|G_1|$. Noting that $G_n = G$ and that $|G_1| \geq 2$, we get $|G| \geq 2^n$. Thus, $n \leq \log_2 |G|$. \square

Returning to the Klein 4-group, we note that it has three sets of irredundant generators: $\{\alpha, \beta\}$, $\{\alpha, \gamma\}$, and $\{\beta, \gamma\}$. It can also be generated by the set $\{\alpha, \beta, \gamma\}$, but this set is redundant since any of its elements can be obtained as the product of the other two.

13.3.4. Permutation Groups

We turn next to an important class of groups, namely groups of permutations.

Definition 13.3.9. (Permutation) A permutation of a set A is a function $\phi : A \rightarrow A$ that is both one to one and onto (a bijection).

Permutations can be “multiplied” using function composition:

$$\sigma\tau(a) \equiv (\sigma \circ \tau)(a) = \sigma(\tau(a))$$

where σ and τ are two permutations of A and $a \in A$. The resulting “product” is also a permutation of A , since it is easily shown to be one to one and onto, and leads us to the following result:

Theorem 13.3.4. (Permutation Group) Given a non-empty set A , let S_A be the set of all permutations of A . Then S_A is a group under permutation multiplication.

Definition 13.3.10. (Symmetric Group) The group of all permutations of the set $\{1, 2, \dots, n\}$ ($n \geq 1$) is the *symmetric group on n letters*, and is denoted by S_n . For any non-empty set T , its symmetric group is denoted by $S_{|T|}(T)$.

Fig. 13.4 shows two example permutations of an 8-element set along with their products. Note that, in general, permutation multiplication is not commutative, i.e., $\sigma\tau \neq \tau\sigma$. The permutations in this figure are presented in a *tabular format* that explicitly shows how each element of the underlying set is mapped to a distinct element of the same set. For instance, element 3 is mapped by σ to 6: $\sigma(3) = 6$. This explicit format becomes cumbersome, however, when the cardinality of the set A is large, and especially when a permutation maps many of the elements of A to themselves. In such cases, the more compact *cycle notation* is preferred. Using cycle notation, the permutations in Fig. 13.4 can be expressed succinctly as follows⁴:

$$\begin{aligned} \sigma &= (1, 8) (3, 6) & \tau &= (2, 7, 8) (3, 5, 6) \\ \sigma\tau &= (1, 8, 2, 7) (3, 5) & \tau\sigma &= (1, 2, 7, 8) (5, 6) \end{aligned}$$

Each such permutation is a product of *disjoint cycles*, where a cycle (a, b, c, \dots, z) is understood to mean that the permutation maps a to b , b to c , and so on, finally

⁴Commas are frequently omitted from cycle notation, especially when the set being permuted consists of single digits. Thus $(2, 7, 8)$ would be written as (278) or $(2\ 7\ 8)$.

$$\sigma = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 8 & 2 & 6 & 4 & 5 & 3 & 7 & 1 \end{bmatrix} \quad \tau = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 7 & 5 & 4 & 6 & 3 & 8 & 2 \end{bmatrix}$$

$$\sigma\tau = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 8 & 7 & 5 & 4 & 3 & 6 & 1 & 2 \end{bmatrix} \quad \tau\sigma = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 2 & 7 & 3 & 4 & 6 & 5 & 8 & 1 \end{bmatrix}$$

Figure 13.4. Example permutations of the set $A = \{1, 2, 3, 4, 5, 6, 7, 8\}$ along with their products.

mapping the last element in the cycle z back to a . An element that does not appear in a cycle is understood to be left fixed (mapped to itself) by the cycle. The *length* of a cycle is the number of elements in the cycle; a cycle with k elements will be referred to as a k -cycle.

Definition 13.3.11. (Permutation Support) Given a permutation σ of a set A , its support consists of those elements of A that are not mapped to themselves by σ :

$$\text{supp}(\sigma) = \{a \in A \mid \sigma(a) \neq a\} \quad (13.6)$$

In other words, $\text{supp}(\sigma)$ are those elements of A that appear in σ 's cycle representation.

Permutation groups are in, some sense, fundamental because of the following result:

Theorem 13.3.5. (Cayley's Theorem) Every group is isomorphic to a group of permutations.

To illustrate, consider the following permutations defined on the Klein 4-group elements:

$$\lambda_\varepsilon = \begin{bmatrix} \varepsilon & \alpha & \beta & \gamma \\ \varepsilon & \alpha & \beta & \gamma \end{bmatrix} \quad \lambda_\alpha = \begin{bmatrix} \varepsilon & \alpha & \beta & \gamma \\ \alpha & \varepsilon & \gamma & \beta \end{bmatrix}$$

$$\lambda_\beta = \begin{bmatrix} \varepsilon & \alpha & \beta & \gamma \\ \beta & \gamma & \varepsilon & \alpha \end{bmatrix} \quad \lambda_\gamma = \begin{bmatrix} \varepsilon & \alpha & \beta & \gamma \\ \gamma & \beta & \alpha & \varepsilon \end{bmatrix}$$

The set of these four permutations along with permutation multiplication as the binary operation on the set satisfy the definition of a group. Additionally, the Klein 4-group is isomorphic to this permutation group using the mapping $\lambda_\varepsilon \leftrightarrow \varepsilon$, $\lambda_\alpha \leftrightarrow \alpha$, $\lambda_\beta \leftrightarrow \beta$, $\lambda_\gamma \leftrightarrow \gamma$.

It is useful at this point to recast the group of negations \mathcal{N}_n introduced earlier as a group of permutations of the set of positive and negative literals on n Boolean variables. Specifically, \mathcal{N}_2 can be expressed as a group of permutations of the set $\{a, a', b, b'\}$. Denoting these permutations by $\eta_\emptyset, \eta_{\{a, a'\}}, \eta_{\{b, b'\}}$ and $\eta_{\{a, a', b, b'\}}$ where η_S indicates that the literals in the set S should be complemented, we can readily write

$$\eta_\emptyset = \begin{bmatrix} a & a' & b & b' \\ a & a' & b & b' \end{bmatrix} \quad \eta_{\{a, a'\}} = \begin{bmatrix} a & a' & b & b' \\ a' & a & b & b' \end{bmatrix}$$

$$\eta_{\{b, b'\}} = \begin{bmatrix} a & a' & b & b' \\ a & a' & b' & b \end{bmatrix} \quad \eta_{\{a, a', b, b'\}} = \begin{bmatrix} a & a' & b & b' \\ a' & a & b' & b \end{bmatrix}$$

We should note that since the literals of each variable are complements of each other, the above permutations can be re-expressed by listing only the positive literals in the top row of each permutation (and by dropping the negative literals from the subscripts of η):

$$\eta_{\emptyset} = \begin{bmatrix} a & b \\ a & b \end{bmatrix}, \eta_{\{a\}} = \begin{bmatrix} a & b \\ a' & b \end{bmatrix}, \eta_{\{b\}} = \begin{bmatrix} a & b \\ a & b' \end{bmatrix}, \eta_{\{a,b\}} = \begin{bmatrix} a & b \\ a' & b' \end{bmatrix} \quad (13.7)$$

These same permutations can be expressed most succinctly in cycle notation as follows:

$$\eta_{\emptyset} = (), \eta_{\{a\}} = (a, a'), \eta_{\{b\}} = (b, b'), \eta_{\{a,b\}} = (a, a')(b, b') \quad (13.8)$$

where the identity permutation is denoted by $()$. In both (13.7) and (13.8) it is implicitly understood that the underlying set whose elements are being permuted is the set of literals $\{a, a', b, b'\}$.

13.3.5. Group of Negations and Permutations

We are now ready to discuss the group of most relevance to our study of symmetry and satisfiability, namely the group of negations *and* permutations of the literals of an n -variable CNF formula. We will denote this group by \mathcal{NP}_n and refer to it as the group of negations and permutations of n variables, with the understanding that its underlying set consists of $2n$ elements that correspond to the literals of n Boolean variables. Any permutation $\pi \in \mathcal{NP}_n$ must satisfy *Boolean consistency* which requires that when literal \hat{x} is mapped to literal \hat{y} , its negated literal \hat{x}' is also simultaneously mapped to literal \hat{y}' , i.e.,

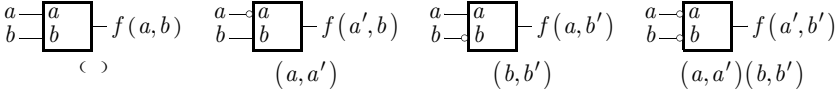
$$\pi(\hat{x}_i') = [\pi(\hat{x}_i)]' \quad (13.9)$$

It is important to note that a group $G \leq S_{2n}$ that permutes the set of $2n$ literals without any restrictions may contain permutations that violate Boolean consistency; e.g. $(a, b')(a', c)$.

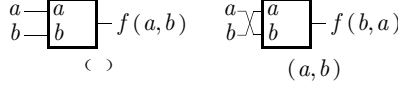
Group \mathcal{NP}_n has two important subgroups: the group of negations of n variables \mathcal{N}_n , and the group of permutations of n variables \mathcal{P}_n ⁵. Fig. 13.5 gives pictorial representations of \mathcal{NP}_2 as well as of its constituent subgroups \mathcal{N}_2 and \mathcal{P}_2 . By convention, when expressing the elements of these groups we will omit the mappings implied by Boolean consistency: thus we will write (a, b') rather than $(a, b')(a', b)$, the latter being implied by the former.

From Fig. 13.5 it is clear that each element of \mathcal{NP}_2 can be expressed as a pair (η, π) where $\eta \in \mathcal{N}_2$ and $\pi \in \mathcal{P}_2$. Furthermore, since both \mathcal{N}_2 and \mathcal{P}_2 are permutation groups, the element $(\eta, \pi) \in \mathcal{NP}_2$ is naturally interpreted as the composition of a negation η followed by a permutation π . For example, $((b, b'), (a, b))$ corresponds to negating b followed by swapping a and b yielding the permutation $(a, b) * (b, b') = (a, b, a', b')$ where the permutation composition operator $*$ is shown explicitly for added clarity.

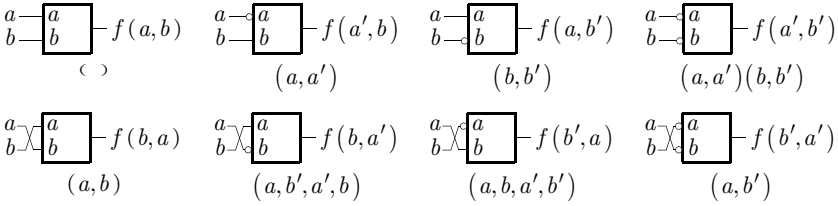
⁵We prefer to denote the group of permutations of n variables by \mathcal{P}_n rather than by S_n to emphasize that its underlying set is the set of $2n$ literals and that its elements obey Boolean consistency (13.9). Other than that, \mathcal{P}_n and S_n are isomorphic groups.



(a) Group \mathcal{N}_2 : negations of two variables



(b) Group \mathcal{P}_2 : permutations of two variables



(c) Group \mathcal{NP}_2 : negations and permutations of two variables

Figure 13.5. Groups of (a) negations, (b) permutations, and (c) mixed negations and permutations of two variables.

More generally, the group \mathcal{NP}_n is the *semi-direct product* [Har63] of \mathcal{N}_n and \mathcal{P}_n , typically denoted by $\mathcal{N}_n \rtimes \mathcal{P}_n$, whose underlying set is the Cartesian product $\mathcal{N}_n \times \mathcal{P}_n = \{(\eta, \pi) | \eta \in \mathcal{N}_n, \pi \in \mathcal{P}_n\}$ and whose group operation $*$ is defined by:

$$(\eta_1, \pi_1) * (\eta_2, \pi_2) = \pi_1 \eta_1 \pi_2 \eta_2 \tag{13.10}$$

For example, $((a, a'), ()) * ((a, a'), (a, b)) = ()(a, a')(a, b)(a, a') = (a, b')$. Clearly, the order of \mathcal{NP}_n is $2^n \cdot n!$.

Returning to the example function in Sec. 13.1, we can express the four transformations that leave it invariant as the semi-direct product group

$$\begin{aligned} & \{(), (a, a')(b, b')\} \rtimes \{(), (a, b)\} \\ &= \{(), (), ((a, a')(b, b'), ()), ((), (a, b)), ((a, a')(b, b'), (a, b))\} \\ &= \{(), (a, a')(b, b'), (a, b), (a, b')\} \end{aligned} \tag{13.11}$$

Note that these four permutations do in fact constitute a group which, again, is easily seen to be isomorphic to the Klein 4-group.

13.3.6. Group Action on a Set

Finally, let's introduce the notion of a group *action* which is fundamental to the development of symmetry-breaking predicates for CNF formulas of Boolean functions.

Table 13.5. Action table of the group $\{(), (a, a')(b, b'), (a, b), (a, b')\}$ on the set $S = \{0, 1, 2, 3, 4, 5, 6, 7\}$ of truth assignments to a, b, c .

	0	1	2	3	4	5	6	7
$\gamma_0 = ()$	0	1	2	3	4	5	6	7
$\gamma_1 = (a, a')(b, b')$	6	7	4	5	2	3	0	1
$\gamma_2 = (a, b)$	0	1	4	5	2	3	6	7
$\gamma_3 = (a, b')$	6	7	2	3	4	5	0	1

Definition 13.3.12. (Group Action) An *action* of a group G on a set S is a map $G \times S \rightarrow S$ such that:

- $es = s$ for all $s \in S$
- $(g_1g_2)(s) = g_1(g_2s)$ for all $s \in S$ and all $g_1, g_2 \in G$.

The group action that we will be primarily concerned with is the action of a group of negations and permutations of the literals of a Boolean function on the set of points (i.e., truth assignments) in the function's domain. For example, Table 13.5 shows the action of the group of negations and permutations in (13.11) on the set of 8 truth assignments to the variables a, b , and c . It is customary to use superscript notation to indicate the action of a group element on an element of the set being acted on. Thus, e.g., $1^{\gamma_1} = 7$. The notation extends naturally to collections (sets, vectors, graphs, etc.), e.g., $\{0, 4, 6, 7\}^{\gamma_3} = \{0^{\gamma_3}, 4^{\gamma_3}, 6^{\gamma_3}, 7^{\gamma_3}\} = \{6, 4, 0, 1\}$.

A group action on a set induces an equivalence partition on the set according to the following theorem:

Theorem 13.3.6. (Group-Induced Equivalence Partition; Orbits) The action of a group G on a set S induces an equivalence relation on the set. Specifically, for $s_1, s_2 \in S$, let $s_1 \sim s_2$ if and only if there exists $g \in G$ such that $gs_1 = s_2$. Then \sim is an equivalence relation on S that partitions it into equivalence classes referred to as the *orbits* of S under G . The orbits of S under a particular group element $g \in G$ are those that satisfy $s_1 \sim s_2$ if and only if $s_2 = g^n s_1$ for some $n \in \mathbb{Z}$.

Using our running example, we can “extract” the orbits of the four permutations in Table 13.5 by inspection:

$$\begin{aligned}
 \text{orbits}(S, \gamma_0) &= \{0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3, 4 \mapsto 4, 5 \mapsto 5, 6 \mapsto 6, 7 \mapsto 7\} \\
 \text{orbits}(S, \gamma_1) &= \{0 \mapsto 6 \mapsto 0, 1 \mapsto 7 \mapsto 1, 2 \mapsto 4 \mapsto 2, 3 \mapsto 5 \mapsto 3\} \\
 \text{orbits}(S, \gamma_2) &= \{0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 4 \mapsto 2, 3 \mapsto 5 \mapsto 3, 6 \mapsto 6, 7 \mapsto 7\} \\
 \text{orbits}(S, \gamma_3) &= \{0 \mapsto 6 \mapsto 0, 1 \mapsto 7 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3, 4 \mapsto 4, 5 \mapsto 5\}
 \end{aligned} \tag{13.12}$$

where $S = \{0, 1, 2, 3, 4, 5, 6, 7\}$ and $x \mapsto y$ means that x is mapped to y by the corresponding permutation. Using the more familiar partition notation, these

same orbits can be equivalently expressed as follows:

$$\begin{aligned}
 \text{partition}(S, \gamma_0) &= \{0|1|2|3|4|5|6|7\} \\
 \text{partition}(S, \gamma_1) &= \{0,6|1,7|2,4|3,5\} \\
 \text{partition}(S, \gamma_2) &= \{0|1|2,4|3,5|6|7\} \\
 \text{partition}(S, \gamma_3) &= \{0,6|1,7|2|3|4|5\}
 \end{aligned} \tag{13.13}$$

The *orbit partition* of S under the group $\{\gamma_0, \gamma_1, \gamma_2, \gamma_3\}$ is now simply the union of the orbits/partitions in (13.12)/(13.13):

$$\begin{aligned}
 \text{orbit partition}(S, \{\gamma_0, \gamma_1, \gamma_2, \gamma_3\}) &= \bigcup_{i \in \{0,1,2,3\}} \text{partition}(S, \gamma_i) \\
 &= \{0,6|1,7|2,4|3,5\}
 \end{aligned} \tag{13.14}$$

Given an element $s \in S$, its orbit under G is the cell in the orbit partition that contains s and is conventionally written as Gs . For example, from (13.14), we get $G0 = \{0,6\}$, $G7 = \{1,7\}$, etc.

A group action on a set distinguishes certain subgroups as *stabilizers*.

Definition 13.3.13. (Stabilizer Subgroup) Given $s \in S$, the subgroup of G that “fixes” s is denoted G_s and defined as $\{\gamma \in G | \gamma s = s\}$. G_s is referred to as the stabilizer of s .

Continuing with the example group in Table 13.5 we note that two of its three non-trivial subgroups are stabilizers. Specifically, $\{\gamma_0, \gamma_2\}$ stabilizes assignments 0, 1, 6, and 7, and can be equivalently denoted as G_0, G_1, G_6 , or G_7 . Similarly, $\{\gamma_0, \gamma_3\}$ stabilizes 2, 3, 4, and 5. The third subgroup $\{\gamma_0, \gamma_1\}$ is not a stabilizer.

13.4. CNF Symmetry

Symmetries of a Boolean function can be classified as being either *semantic* or *syntactic*. Semantic symmetries are intrinsic properties of the function that are independent of any particular representation we may choose for it. In contrast, syntactic symmetries correspond to specific algebraic representations of the function, e.g., an SOP, POS, or some nested expression that might correspond to a circuit representation. Our focus will be on the CNF representation of a function since it is the usual form that is processed by SAT solvers. We should note, however, that a given function can be represented by many equivalent CNF formulas. In particular, some of these formulas are unique representations and can, thus, be used to identify a function’s semantic symmetries. Examples of such unique CNF forms include the set of a function’s maxterms, the complete set of its prime implicants, or (if it exists) its unique minimal CNF expression. Such representations, however, are generally exponential in the number of variables rendering any attempt to analyze them intractable.

As mentioned in Sec. 13.3.5, we consider the invariance of an n -variable CNF formula $\varphi(X)$ under a) negation, b) permutation, and c) combined negation and permutation of its variables. We will refer to the symmetries that arise under

Table 13.6. Symmetries of different CNF formulas for the example function in (13.1). The formula in row 1 consists of the function’s maxterms, whereas that in row 2 is the function’s complete set of prime implicants. These two forms are unique and will always yield the function’s semantic symmetries. Other unique CNF representations, such as a unique minimal CNF form (which in the case of this function is the same as the complete set of prime implicants) will also yield the group of semantic symmetries. The remaining formulas will, generally, yield syntactic symmetries which are subgroups of the group of semantic symmetries.

(Equivalent) CNF Formulas	Symmetries			
	Identity ()	Variable (a, b)	Value (a, a')(b, b')	Mixed (a, b')
1. $(a + b + c)(a + b + c')(a + b' + c)$ $(a' + b' + c)(a' + b' + c')(a' + b + c)$	✓	✓	✓	✓
2. $(a + b)(a' + b')(c)$	✓	✓	✓	✓
3. $(a + b)(a' + b')(a' + c)(a + c)$	✓	✓	✓	✓
4. $(a + b)(a' + b' + c')(c)$	✓	✓		
5. $(a + c)(a' + c)(a' + b' + c')(a + b + c')$	✓		✓	
6. $(a + b)(a' + b')(a + b' + c)(c)$	✓			✓
7. $(a + b)(b' + c)(a' + b' + c)(c)$	✓			

these three types of transformation as *value*, *variable*, and *mixed* symmetries, respectively, and denote the resulting symmetry group by G_φ . Clearly, G_φ is a subgroup of \mathcal{NP}_n , i.e., $G_\varphi \leq \mathcal{N}_n \times \mathcal{P}_n$. The action of G_φ on the formula’s literals results in a re-ordering of the formula’s clauses and of the literals within the clauses while preserving Boolean consistency (13.9).

While it may be obvious, it is useful to demonstrate how different CNF formulas for the same function can have drastically different symmetries, or even no symmetries other than the identity. Table 13.6 gives the symmetries of seven different formulas for our example function (13.1). Clearly, a good *encoding* is necessary if we are to reap the benefits of symmetry detection and breaking.

13.5. Automorphism Group of a Colored Graph

So far we have been concerned with what groups are and what properties they have. Here we introduce one last group, the automorphism group of a colored graph, and sketch the basic procedure for computing it. As we show in Sec. 13.6, the symmetries of a CNF formula can be derived by computing the automorphism group of an associated graph. Efficient algorithms for graph automorphism are, thus, a critical link in the overall symmetry detection and breaking flow.

Definition 13.5.1. (Automorphism Group [McK81]) Given a graph⁶ $G = (V, E)$, with vertex set $V = \{1, 2, \dots, n\}$ and edge set E , along with a partition $\pi(V) = \{V_1 | V_2 | \dots | V_k\}$ of its vertices, its *automorphism group* $\text{Aut}(G, \pi)$ is $\{\gamma \in S_n | G^\gamma = G \text{ and } \pi^\gamma = \pi\}$, where $G^\gamma = (V^\gamma, E^\gamma)$ with $E^\gamma = \{(u^\gamma, v^\gamma) | (u, v) \in E\}$ and $\pi^\gamma = \{V_1^\gamma, \dots, V_k^\gamma\}$.

In other words, $\text{Aut}(G, \pi)$ is the set of permutations of the graph vertices that map edges to edges and non-edges to non-edges with the restriction that vertices

⁶The use of G and π in this definition to denote *graph* and *partition*, respectively, should not be confused with their earlier use to denote *group* and *permutation*.

in any given cell of π can only be mapped to vertices in that same cell. It is customary to consider the partition $\pi(V)$ as an assignment of k different colors to the vertices and to view it as a constraint that disallows permutations that map vertices of some color to vertices of a different color. We will be principally interested in *ordered partitions*, also called *colorings*, of the vertices. Thus, $\pi(V)$ will be specified with square, rather than curly, brackets: $\pi(V) = [V_1 | V_2 | \dots | V_k]$.

The importance of the graph automorphism problem in the context of symmetry detection for CNF formulas stems from the fact that it can be solved fairly efficiently for a large class of graphs. In fact, while this problem is known to be in the complexity class NP [GJ79], it is still an open question whether it is NP-complete or is in P. A straightforward, but hopelessly impractical, procedure for computing $\text{Aut}(G, \pi)$ is to explicitly enumerate all permutations in S_n and to discard those that are not symmetries of G . Efficient graph automorphism algorithms employ more intelligent implicit enumeration procedures that drastically reduce the number of individual permutations that must be examined. The kernel computation in such algorithms is an *ordered partition refinement* procedure similar to that used for state minimization of finite-state automata [AHU74], and their underlying data structure is a search tree whose nodes represent colorings of the graph vertices.

Definition 13.5.2. (Color-Relative Vertex Degree) Given a coloring $\pi(V) = [V_1 | V_2 | \dots | V_k]$ of the graph G and a vertex $v \in V$, let $d(v, V_i)$ be the number of vertices in V_i that are adjacent to v in G . Note that $d(v, V)$ is simply the degree of v in G .

Definition 13.5.3. (Stable/Equitable Coloring) A coloring π is *stable* (also referred to as equitable) if

$$d(u, V_i) = d(v, V_i), \quad 1 \leq i \leq |\pi|$$

for all pairs of vertices $u, v \in V_j, 1 \leq j \leq |\pi|$.

Ordered partition refinement transforms a given initial coloring π into a final *coarsest* stable coloring $\pi' \leq \pi$ by repeatedly splitting cells containing vertices with different vertex degrees (relative to the current coloring). Fig. 13.6 illustrates this procedure on a sample 2-colored graph with 9 vertices. The initial coloring π consists of two cells corresponding to the graph's two types of vertices (square and round). In the first refinement iteration, vertex 3 is split off from V_1 because its degree relative to V_2 is different from those of the two other vertices in V_1 . In other words, vertex 3 can be *distinguished* from vertices 1 and 2 and cannot possibly be mapped to either by a graph symmetry. Similarly, vertex 9 is split off from V_2 yielding the intermediate coloring $\hat{\pi}$. The second refinement iteration splits vertex 8 from V_3 yielding the final stable coloring π' .

If the refinement procedure returns a discrete coloring π' , i.e. every cell of the partition is a singleton, then all vertices can be distinguished, implying that G has no symmetries besides the identity. However, if π' is not discrete, then there is some non-singleton cell in π' representing vertices that could not be distinguished based on degree and are, thus, candidates for some symmetry. This is checked by

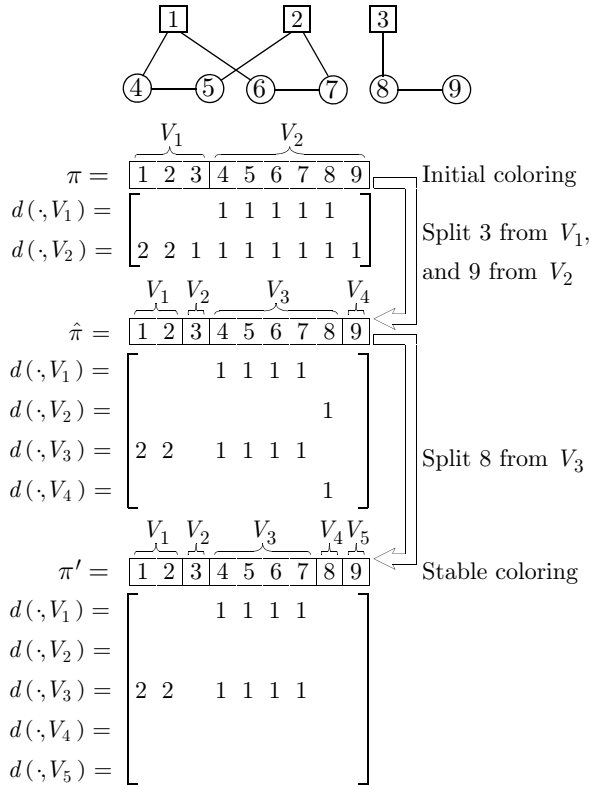


Figure 13.6. Illustration of ordered partition refinement. The “matrix” of vertex degrees is indicated under each coloring (zero entries are shown as blanks, and the degrees of singleton cells are not shown). Efficient implementations of this procedure compute only those vertex degrees necessary to trigger refinement.

selecting some non-singleton cell T of π' , called the *target cell*, and forming $|T|$ descendant colorings from π' , each identical to π' except that a distinct $t \in T$ is placed in front of $T - \{t\}$. Each of these colorings is subsequently refined, and further descendant colorings are generated if the refined colorings are not discrete; this process is iterated until discrete colorings are reached. The colorings explored in this fashion form a *search tree* with the discrete colorings at the leaves.

Fig.13.7 illustrates this process for an example 6-vertex graph. The initial coloring has a single cell since all vertices have the same color. Refinement yields a stable coloring with two non-singleton cells. At this point, the first cell is chosen as a target, and we branch by creating three descendant colorings. The process is now repeated (i.e., we refine each derived coloring and branch from it if it has non-singleton cells) in a depth-first manner until discrete colorings are reached. In this example, the search tree terminates in six leaves corresponding to six different discrete colorings.

The next step in the process is to derive from these colorings permutations

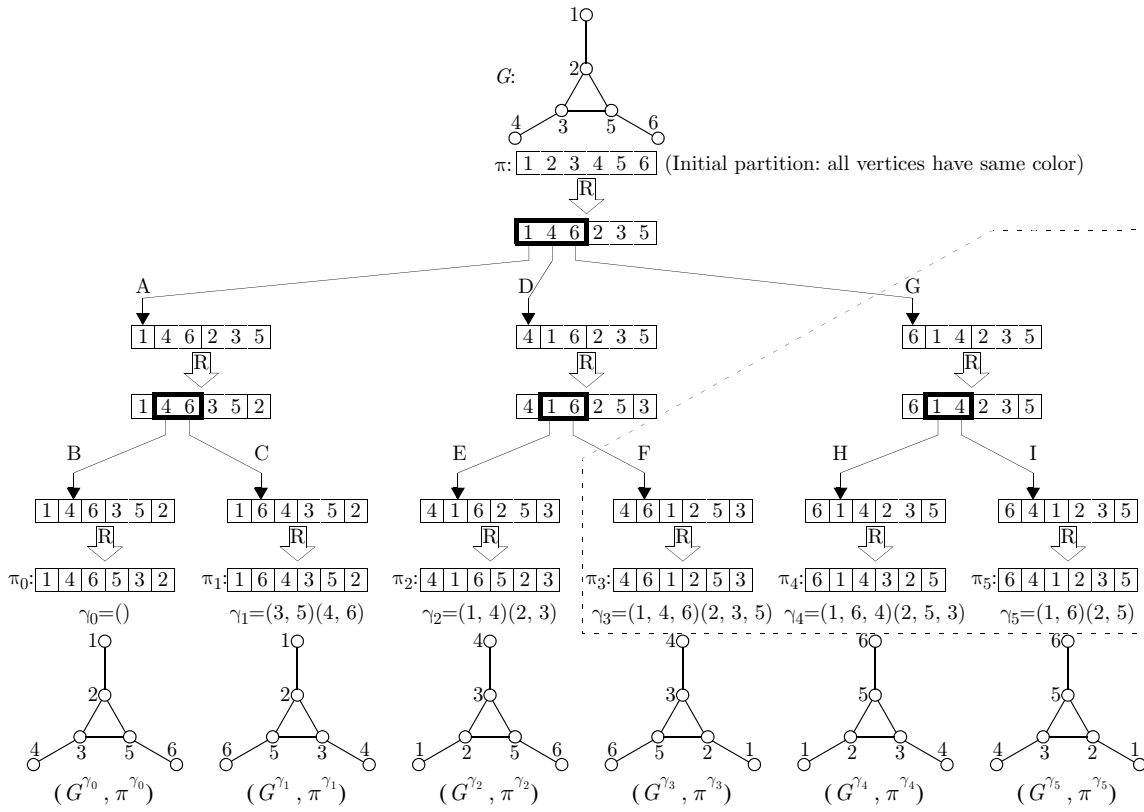


Figure 13.7. Basic flow of the ordered partition refinement procedure for computing the automorphism group of an example 6-vertex single-colored graph. Each node in the search tree is a sequence of ordered partitions (obtained using the partition refinement procedure illustrated in Fig. 13.6 and indicated by the fat arrows labeled with ‘R’) ending with a stable coloring. The tree is traversed in the order A-B-C-D-E-F-G-H-I.

Table 13.7. Automorphism results on a sample of very large sparse graphs (sorted in descending order by number of vertices) using a modern graph automorphism tool. The *bigblue* and *adaptec* graphs are derived from circuits used in the ISPD 2005 placement competition [ISP]. The *CA* and *FL* graphs represent, respectively, the road networks of California and Florida [U.S]. The *Internet* graph represents the interconnections of the major routers on the Internet's North American backbone [CBB00, GT00]. Note the extreme sparsity of each of these graphs and of their symmetry generators. Note also the astronomical orders of the graphs' automorphism groups. These data are from [DSM08].

Name	Graph $G = (V, E)$			Symmetry Group $\text{Aut}(G)$			Time (s)
	$ V $	$ E $	Avg. Deg.	$ \text{Aut}(G) $	Generators	Avg. Supp.	
bigblue4	3,822,980	8,731,076	4.568	10^{119182}	215,132	2.219	5.361
bigblue3	1,876,918	3,812,614	4.063	10^{52859}	98,567	2.129	2.337
CA	1,679,418	2,073,394	2.469	10^{14376}	44,439	2.280	0.838
FL	1,063,465	1,329,206	2.500	10^{13872}	33,225	2.466	0.543
adaptec4	878,800	1,880,109	4.279	10^{24443}	53,857	2.261	0.989
adaptec3	800,506	1,846,242	4.613	10^{15450}	36,289	2.157	0.929
Internet	284,805	428,624	3.010	10^{83687}	108,743	2.281	0.405

of the graph vertices that correspond to graph symmetries. This is done by choosing one of these colorings as a reference, and computing the permutations that transform it to the other colorings. For the example of Fig.13.7, we chose the left-most coloring π_0 as the reference; the permutations that convert it to the other colorings, including itself, are labeled γ_0 to γ_5 . Each such permutation γ_i is now checked to determine if it is a graph symmetry, i.e., if $G^{\gamma_i} = G$. In this example all six permutations are indeed symmetries of the graph, yielding $\text{Aut}(G) = \{\gamma_0, \gamma_1, \gamma_2, \gamma_3, \gamma_4, \gamma_5\}$. A permutation that does not correspond to a symmetry of the graph triggers backtracking in the search tree so that other branches can be explored.

While the above sketch of graph automorphism algorithms is sufficient for a basic understanding of how they operate, we should note that their performance in practice depends critically on aggressive pruning of the search tree to avoid deriving permutations that can be obtained as the product of other permutations that have already been found. For our running example, the portion of the search tree enclosed by the dashed outline (nodes F, G, H, and I) can be safely pruned away since the permutations identified at its leaves can be expressed in terms of permutations γ_1 and γ_2 . Ideally, the output of a graph automorphism algorithm should be a *minimal* set of irredundant generators. In practice, however, the overhead of performing the group-theoretic pruning necessary to guarantee this outcome tends to be excessive. Instead, graph automorphism tools are designed to produce at most $n - 1$ generators for an input graph with n vertices, providing an exponentially smaller but not necessarily minimal representation of the complete set of symmetries.

Modern implementations of graph automorphism can process graphs with millions of vertices in a matter of seconds on commodity personal computers. Key to this remarkable performance is the extreme *sparsity* of both the graphs themselves and the generators of their symmetry groups. In this context, sparsity is taken to mean that the average vertex degree in the graph and the average

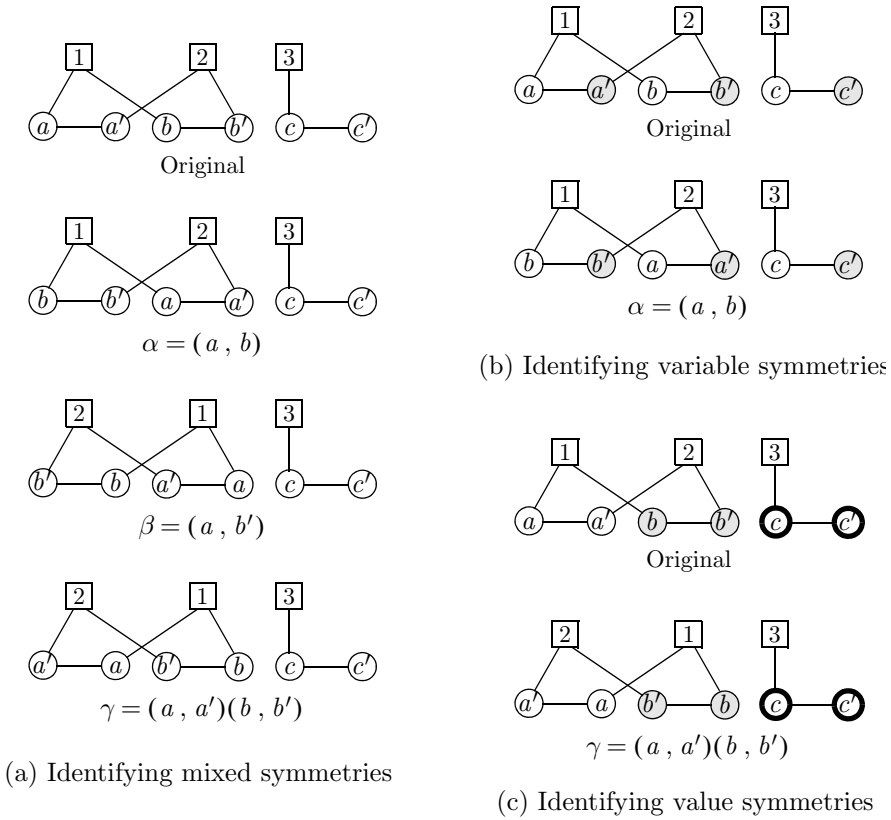


Figure 13.8. Colored graph constructions and resultant symmetries for the example formula $(a + b)(a' + b')(c)$. (a) Using two colors (square vertices: color 0; circle vertices: color 1) yields the symmetry group $G = \{(), (a, b), (a, b'), (a, a')(b, b')\}$. (b) Using three colors (shaded circles: color 2) yields the symmetry group $G = \{(), (a, b)\}$. (c) Using four colors (bolded circles: color 3) yields the symmetry group $G = \{(), (a, a')(b, b')\}$.

support of the resulting symmetry generators are both much smaller than the number of graph vertices. Incorporating knowledge of both types of sparsity in the basic automorphism algorithm can result in substantial pruning of the search tree, essentially yielding a tree whose size is linear, rather than quadratic, in the total support of the symmetry generators [DSM08]. Table 13.7 provides empirical evidence of these observations on a number of very large sparse graphs.

13.6. Detection of CNF Symmetries

The basic approach for identifying the symmetries of a CNF formula is through reduction to, and solution of, an associated colored graph automorphism problem. Specifically, given a CNF formula consisting of m clauses and l (total) literals over n variables, construct a corresponding colored graph with $m + 2n$ vertices and

$l + n$ edges as follows:

- **Clause vertices:** represent each of the m clauses by a vertex of color 0
- **Positive literal vertices:** represent each of the n positive literals by a vertex of color 1
- **Negative literal vertices:** represent each of the n negative literals by a vertex of color 1
- **Clause edges:** represent each of the m clauses by a set of edges that connect the clause vertex to the vertices of the literals that appear in the clause
- **Boolean consistency edges:** connect each pair of literal vertices that correspond to the same variable by an edge

The choice of two vertex colors insures that clause vertices can only be mapped to clause vertices and literal vertices can only be mapped to literal vertices. Using one color for all literal vertices also makes it possible to detect the mixed symmetries of the formula. Pure variable and pure value symmetries can be identified by using more colors for the literal vertices. Specifically, variable symmetries can be identified by using two colors: color 1 for positive literal vertices and color 2 for negative literal vertices. Identifying pure value symmetries requires n colors: color 1 for the literals of the first variable, color 2 for the literals of the second variable, and so on. This construction, thus, is general enough to identify variable, value, and mixed symmetries by a suitable assignment of colors to literal vertices. Fig. 13.8 illustrates these variants, along with the symmetries they yield, for the three-variable CNF formula $(a + b)(a' + b')(c)$ of our example function in (13.1).

Noting that graph automorphism algorithms tend to be more sensitive to the number of vertices of an input graph than to the number of its edges, the above construction can be optimized to reduce the number of graph vertices while preserving its automorphism group. Specifically, single-literal clauses can be eliminated through Boolean Constraint Propagation (BCP) yielding an equivalent formula that has only clauses with 2 or more literals each. A further reduction in the number of graph vertices is achieved by modeling binary clauses (i.e., 2-literal clauses) using graph edges instead of graph vertices. As we demonstrate shortly, the CNF formulas used to model problems from a variety of application domains tend to have a large fraction of binary clauses. Thus, this optimization is particularly significant for the overall efficiency of symmetry detection. Care must be taken, however, when applying this optimization since it may, on rare occasions, lead to spurious symmetries that result from unintended mappings between binary clause edges and Boolean consistency edges. Such symmetries violate the Boolean consistency requirement (13.9) and, as shown in [ARMS03], arise when the CNF formula contains chains of implications such as $(x'_1 + x_2)(x'_2 + x_3) \cdots (x'_k + x_1)$. The set of generators returned by a graph automorphism program must, thus, be checked for members that violate (13.9) before reporting the results back. Several options of how to filter these spurious symmetries are described in [ARMS03].

The results of applying the above graph construction to detect the symmetries of a selection of CNF instances are given in Table 13.8. These instances are drawn from the following six benchmark families.

- **Hole:** This is the family of unsatisfiable pigeon-hole instances. Instance ‘hole n ’ corresponds to placing $n + 1$ pigeons in n holes [DIM].
- Difficult CNF instances that represent the routing of wires in integrated circuits [ARMS02, ARMS03]. This family includes three members: the first two share the characteristic of the pigeon-hole family (namely placing n objects in m slots such that each object is in a slot and no two objects are in the same slot) whereas the third member is based on the notion of *randomized flooding*:
 - **ChnlRoute:** This is a family of unsatisfiable channel routing instances. Instance ‘chnl n _ m ’ models the routing of m wires through n tracks in a routing channel.
 - **FPGARoute:** This is a family of satisfiable instances that model the routing of wires in the channels of field-programmable integrated circuits [NASR01]. Instance ‘fpga n _ m ’ models the routing of m wires in channels of n tracks.
 - **GRoute:** This is a family of randomly-generated satisfiable instances that model the routing of global wires in integrated circuits. Instance ‘sn- m - c - k ’ corresponds to global routing of wires on an n -by- m grid with *edge capacity* between grid cells equal to c tracks. The last integer k in the instance name serves as the instance identifier.
- **Urq:** This is the family of unsatisfiable randomized instances based on expander graphs [Urq87].
- **XOR:** This is a family of various *exclusive or* chains from the SAT’02 competition [SAT].
- **Pipe:** This is a family of difficult unsatisfiable instances that model the functional correctness requirements of modern out-of-order microprocessor CPUs [VB01].

The results in Table 13.8 were generated by a) constructing three colored graphs for each CNF instance and, b) using the **saucy** graph automorphism program [DLSM04] to obtain corresponding sets of generators for the graphs’ automorphism groups. These generators were then postprocessed (by dropping cycles that permute clause vertices, and by relabeling negative literal vertices to relate them back to their positive literals) to obtain the symmetry generators of the CNF instance. The generator sets corresponding to the three graph variants identify, respectively, a formula’s mixed, variable, and value symmetries. The experiments were conducted on a 3GHz Intel Xeon workstation with 4GB RAM running the Redhat Linux Enterprise operating system.

For each benchmark instance, the table lists the following data:

- Instance parameters: family, name, number of variables, total number of clauses, number of binary clauses, number of literals, and number of vertices in the corresponding colored graph.
- Symmetry group characteristics for each of the three variants of the colored graph: group order, number of generators along with their total support, and symmetry detection time.

Examination of the data in Table 13.8 leads to the following general observations:

- Except for the Pipe microprocessor verification benchmarks which only have a handful of symmetries, the number of symmetries in these instances is extremely large.
- Symmetry detection is very fast, in most cases taking only tiny fractions of a second. The notable exceptions are the GRoute benchmarks and the larger Urq benchmarks which require on the order of a few seconds. This is partly explained by the large graphs used to model them which, unlike those of the other instances, have thousands rather than hundreds of vertices. On the other hand, the fast detection times for the Pipe instances, despite their large graphs, can be attributed to the small number of symmetries in those graphs.
- While **saucy** guarantees that the maximum number of generators it outputs will be no more than the number of graph vertices minus one, in all cases the number of produced generators is much less. In fact, the logarithmic upper bound given in Theorem 13.3.3 for the size of an irredundant set of generators is reached only for the Urq and Pipe benchmarks. The number of generators in other cases is less than this bound, sometimes by a large factor.
- The Hole and wire routing benchmarks exhibit only pure variable symmetries. This is to be expected as these benchmarks essentially involve some sort of capacity constraint on a set of interchangeable variables. Note also that the two varieties of colored graphs corresponding to mixed and variable symmetries yield the same number of generators in all cases.
- The Urq and XOR benchmarks, except for x1_16, exhibit only pure value symmetries. Again the graphs for mixed and value symmetries yield the same number of generators in those cases. But note that the support of these generators is different suggesting that **saucy** is producing different sets of mixed and value symmetry generators. The x1_16 instance is the only one in the entire set of benchmarks in the table to have all types of symmetry: 2 variable permutations, $6.55E+4$ variable negations, and their composition.
- The Pipe benchmarks do not have any value symmetries, but do have variable symmetries and mixed symmetries that involve simultaneous variable permutations and negations.
- Finally, the support of the symmetry generators in all cases is a very small fraction of the total number of variables, ranging from 4% to 29% with a mean of 13%. In other words, the generators are extremely sparse.

We should note that these benchmarks are relatively small and that it is quite conceivable that other benchmark families might yield a different set of results. However, it seems safe to assume that the detection of the symmetries of CNF formulas through reduction to graph automorphism is computationally quite feasible using today's graph automorphism programs, and that the use of such programs as pre-processors for SAT solvers can be a viable approach in many cases.

Table 13.9. Symmetries of the formula $\varphi(a, b, c) = (a + b' + c')(a + b + c)(a' + b + c')$ along with their orbits and lex-leader predicates (the lex-leaders in each orbit are shown in bold.) Note that we have omitted the identity permutation π_0 since each of the 8 truth assignments will be in its own orbit yielding a lex-leader predicate which is the constant 1 function.

Permutation	Orbits	Lex-Leader Predicate
$\pi_1 = (a, b)$	$\{\mathbf{0} \mapsto 0, \mathbf{1} \mapsto 1, \mathbf{2} \mapsto 4 \mapsto 2, \mathbf{3} \mapsto 5 \mapsto 3, \mathbf{6} \mapsto 6, \mathbf{7} \mapsto 7\}$	$\sum_{(a,b,c)} (0, 1, 2, 3, 6, 7) = (a' + b)$
$\pi_2 = (a, c')$	$\{\mathbf{0} \mapsto 5 \mapsto 0, \mathbf{1} \mapsto 1, \mathbf{2} \mapsto 7 \mapsto 2, \mathbf{3} \mapsto 3, \mathbf{4} \mapsto 4, \mathbf{6} \mapsto 6\}$	$\sum_{(a,b,c)} (0, 1, 2, 3, 4, 6) = (a' + c')$
$\pi_3 = (b, c')$	$\{\mathbf{0} \mapsto 3 \mapsto 0, \mathbf{1} \mapsto 1, \mathbf{2} \mapsto 2, \mathbf{4} \mapsto 7 \mapsto 4, \mathbf{5} \mapsto 5, \mathbf{6} \mapsto 6\}$	$\sum_{(a,b,c)} (0, 1, 2, 4, 5, 6) = (b' + c')$
$\pi_4 = (a, b, c')$	$\{\mathbf{0} \mapsto \mathbf{3} \mapsto 5 \mapsto 0, \mathbf{1} \mapsto 1, \mathbf{2} \mapsto 7 \mapsto 4 \mapsto 2, \mathbf{6} \mapsto 6\}$	$\sum_{(a,b,c)} (0, 1, 2, 3, 6) = (a' + b)(a' + c')$
$\pi_5 = (a, c', b)$	$\{\mathbf{0} \mapsto 5 \mapsto 3 \mapsto 0, \mathbf{1} \mapsto 1, \mathbf{2} \mapsto \mathbf{4} \mapsto 7 \mapsto 2, \mathbf{6} \mapsto 6\}$	$\sum_{(a,b,c)} (0, 1, 2, 4, 6) = (a' + c')(b' + c')$
Group	Equivalence Classes	Lex-Leader Predicate
$\{\pi_0, \pi_1, \pi_2, \pi_3, \pi_4, \pi_5\}$	$\{\mathbf{0}, \mathbf{3}, \mathbf{5} \mid \mathbf{1} \mid \mathbf{2}, \mathbf{4}, \mathbf{7} \mid \mathbf{6}\}$	$\sum_{(a,b,c)} (0, 1, 2, 6) = (a' + b)(b' + c')$

13.7. Symmetry Breaking

Given a CNF formula $\varphi(X)$ along with its group of symmetries G_φ the goal now is to augment $\varphi(X)$ with additional predicates that break all of its symmetries. Recall that G_φ induces an equivalence relation on the set of truth assignments, i.e., it partitions the space of possible assignments into equivalence classes. Complete symmetry breaking amounts to constructing predicates that select exactly one representative assignment from each equivalence class. The most common approach for accomplishing this objective is to order the assignments numerically, and to use the *leq* predicate introduced in Sec. 13.2 to choose the least assignment in each equivalence class. Specifically, let $\text{PP}(\pi; X)$ denote the *permutation predicate* of permutation $\pi \in G_\varphi$ defined as:

$$\begin{aligned} \text{PP}(\pi; X) &= \text{leq}(X, X^\pi) \\ &= \bigwedge_{i=0}^{n-1} \left[\bigwedge_{j=i+1}^{n-1} (x_j = x_j^\pi) \right] \rightarrow (x_i \leq x_i^\pi) \end{aligned} \quad (13.15)$$

$\text{PP}(\pi; X)$ is a Boolean function that evaluates to 1 for each assignment X^* such that $X^* \leq \pi(X^*)$ and evaluates to 0 otherwise. More specifically, let $X_0^* \mapsto X_1^* \mapsto \dots \mapsto X_{k-1}^* \mapsto X_0^*$ be an orbit of π , i.e. π maps each assignment X_i^* to the next assignment $X_{(i+1) \bmod k}^*$; then $\text{PP}(\pi; X)$ is true for only those assignments that are numerically smaller than their successor assignments as we traverse the orbit. In particular, $\text{PP}(\pi; X)$ will be true for the smallest assignment in the orbit. We will refer to $\text{PP}(\pi; X)$ as a *lex-leader predicate* for permutation π . We can now construct a Boolean predicate that breaks *every* symmetry in G_φ by conjoining all of its permutation predicates:

$$\rho(G_\varphi; X) = \bigwedge_{\pi \in G_\varphi} \text{PP}(\pi; X) \quad (13.16)$$

This symmetry-breaking predicate (SBP) will be true for exactly the least assignment in each equivalence class induced by G_φ on the set of truth assignments. An example illustrating the construction of this predicate is shown in Table 13.9.

Using the SBP in (13.16), we can now determine the satisfiability of $\varphi(X)$ by invoking a SAT solver on the formula

$$\psi(X) \equiv \varphi(X) \wedge \rho(G_\varphi; X) \quad (13.17)$$

Clearly, if $\varphi(X)$ is unsatisfiable, then so is $\psi(X)$. On the other hand, if $\varphi(X)$ is satisfiable then its restriction by $\rho(G_\varphi; X)$ preserves only those solutions that are *numerically least* in their orbits. Either way, we can determine the satisfiability of $\varphi(X)$ by, instead, checking the satisfiability of $\psi(X)$.

To apply a SAT solver to (13.17), the permutation predicates in (13.15) must first be converted to CNF. A simple way to accomplish this is to introduce n auxiliary *equality* variables

$$e_i \equiv (x_i = x_i^\pi), \quad 0 \leq i \leq n-1 \quad (13.18)$$

allowing us to re-write (13.15) as

$$\text{PP}(\pi; X) = \bigwedge_{i \in [0, n-1]} (e'_{i+1} + e'_{i+2} + \cdots + e'_{n-1} + x'_i + x_i^\pi) \quad (13.19)$$

This leads to a CNF representation of the permutation predicate whose size is

$$\begin{aligned} \text{clauses}(\text{PP}(\pi; X)) &= 5n \\ \text{literals}(\text{PP}(\pi; X)) &= 0.5(n^2 + 27n) \end{aligned} \quad (13.20)$$

Table 13.10. Comparison of CNF formula size and permutation predicate size from (13.20)

		Benchmark			Permutation Predicate		
Family	Name	Vars	Clauses	Lits	Vars	Clauses	Lits
Hole	hole7	56	204	448	56	280	2324
	hole8	72	297	648	72	360	3564
	hole9	90	415	900	90	450	5265
	hole10	110	561	1210	110	550	7535
	hole11	132	738	1584	132	660	10494
	hole12	156	949	2028	156	780	14274
ChnlRoute	chnl10_11	220	1122	2420	220	1100	27170
	chnl10_12	240	1344	2880	240	1200	32040
	chnl10_13	260	1586	3380	260	1300	37310
	chnl11_12	264	1476	3168	264	1320	38412
	chnl11_13	286	1742	3718	286	1430	44759
	chnl11_20	440	4220	8800	440	2200	102740
FPGARoute	fpga10_8	120	448	1200	120	600	8820
	fpga10_9	135	549	1485	135	675	10935
	fpga12_8	144	560	1488	144	720	12312
	fpga12_9	162	684	1836	162	810	15309
	fpga12_11	198	968	2640	198	990	22275
	fpga12_12	216	1128	3096	216	1080	26244
	fpga13_9	176	759	2031	176	880	17864
	fpga13_10	195	905	2440	195	975	21645
	fpga13_12	234	1242	3396	234	1170	30537
Groute	s3-3-3-1	864	7592	16038	864	4320	384912
	s3-3-3-3	960	9156	19258	960	4800	473760
	s3-3-3-4	912	8356	17612	912	4560	428184
	s3-3-3-8	912	8356	17612	912	4560	428184
	s3-3-3-10	1056	10862	22742	1056	5280	571824
Urq	Urq3_5	46	470	2912	46	230	1679
	Urq4_5	74	694	4272	74	370	3737
	Urq5_5	121	1210	7548	121	605	8954
	Urq6_5	180	1756	10776	180	900	18630
	Urq7_5	240	2194	13196	240	1200	32040
	Urq8_5	327	3252	20004	327	1635	57879
XOR	x1_16	46	122	364	46	230	1679
	x1_24	70	186	556	70	350	3395
	x1_32	94	250	748	94	470	5687
	x1_36	106	282	844	106	530	7049
Pipe	2pipe_1_000	834	7026	19768	834	4170	359037
	2pipe_2_000	925	8213	23161	925	4625	440300
	2pipe	861	6695	18637	861	4305	382284

The size of a single permutation predicate can, thus, significantly exceed the size of the original formula! Table 13.10 compares the size of each benchmark in Table 13.8 to the size of a single permutation predicate as formulated in (13.18) and (13.19). In practice, this dramatic increase in the size of the formula negates much of the hoped-for reduction in the search space due to symmetry breaking. This is further compounded by the fact that full symmetry breaking requires adding, according to (13.16), a quadratically-sized predicate *for each permutation in the symmetry group*. As the data in Table 13.8 clearly demonstrate, the orders of the symmetry groups for most benchmarks are exponential in the number of variables, rendering direct application of (13.18)-(13.19) infeasible.

This analysis may suggest that we have reached an impasse: the theoretical possibility of significantly pruning the search space by breaking a formula's symmetries seems to be at odds with the need to add an exponentially-sized SBP to the formula causing a SAT solver to grind to a halt. We show next a practical approach to resolve this impasse and demonstrate its effectiveness on the benchmark families analyzed in Sec. 13.6. The approach rests on two key ideas: a) reducing the size of the permutation predicate so that it becomes sub-linear, rather than quadratic, in the number of variables, and b) relaxing the requirement to break all symmetries and opting, instead, to break "enough" symmetries to reduce SAT search time.

13.7.1. Efficient Formulation of the Permutation Predicate

To obtain an efficient CNF representation of the permutation predicate in (13.15) it is necessary to introduce some additional notation that facilitates the manipulation of various subsets of the variables $\{x_i | 0 \leq i \leq n - 1\}$. Let I_n denote the integer set $\{0, 1, \dots, n - 1\}$, and let upper-case "index variables" I and J denote non-empty subsets of I_n as appropriate. Given an index set I and an index $i \in I$, define the *index selector* functions:

$$\begin{aligned} \text{pred}(i, I) &= \{j \in I \mid j < i\} \\ \text{succ}(i, I) &= \{j \in I \mid j > i\} \end{aligned} \tag{13.21}$$

A permutation is said to be a *phase-shift* permutation if it contains one or more 2-cycles that have the form (x_i, x'_i) , i.e., cycles that map a variable to its complement. Such cycles will be referred to as *phase-shift cycles*. Given a permutation π , we now define

$$\text{ends}(\pi) = \{i \in I_n \mid i \text{ is the smallest index of a literal in a non-phase-shift cycle of } \pi\} \tag{13.22}$$

and

$$\text{phase-shift}(\pi) = \max \{i \in I_n \mid x_i^\pi = x'_i\} \tag{13.23}$$

In other words, based on the assumed total ordering $x_{n-1} \succ x_{n-2} \succ \dots \succ x_0$, $\text{ends}(\pi)$ identifies the *last* literal in each non-phase-shift cycle of π whereas $\text{phase-shift}(\pi)$ identifies the literal of the *first* phase-shift cycle.

We begin the simplification of the permutation predicate in (13.15) by rewriting it as a conjunction of *bit predicates*:

$$\text{BP}(\pi; x_i) = \left[\bigwedge_{j \in \text{succ}(i, I_n)} (x_j = x_j^\pi) \right] \rightarrow (x_i \leq x_i^\pi) \quad (13.24)$$

$$\text{PP}(\pi; X) = \bigwedge_{i \in I_n} \text{BP}(\pi, x_i) \quad (13.25)$$

Minimization of the CNF representation of the permutation predicate is based on two key optimizations. The first utilizes the *cycle structure* of a permutation to eliminate tautologous bit predicates and is accomplished by replacing I_n in (13.24) and (13.25) with $I \subset I_n$ where $|I| \ll n$. This optimization can be viewed as replacing n in (13.20) by a much smaller number m . The second optimization decomposes the multi-way conjunction in (13.24) into a chain of 2-way conjunctions to yield a CNF formula whose size is linear, rather than quadratic, in m . Fig. 13.9 provides an example illustrating these optimizations.

Elimination of redundant BPs. Careful analysis of (13.24) reveals three cases in which a BP is tautologous, and hence redundant. The first corresponds to bits that are mapped to themselves by the permutation, i.e., $x_i^\pi = x_i$. This makes the consequent of the implication in (13.24), and hence the whole bit predicate, unconditionally true. With a slight abuse of notation, removal of such BPs is easily accomplished by replacing the index set I_n in (13.24) and (13.25) with $\text{supp}(\pi)$ ⁷. For sparse permutations, i.e., permutations for which $|\text{supp}(\pi)| \ll n$, this change alone can account for most of the reduction in the CNF size of the PP.

The second case corresponds to the BP of the last bit in each non-phase-shift cycle of π . “Last” here refers to the assumed total ordering on the variables. Assume a cycle involving the variables $\{x_j | j \in J\}$ for some index set J and let $e = \min(J)$. Such a cycle can always be written as (x_s, \dots, x_e) where $s \in J$. Using equality propagation, the portion of the antecedent in (13.24) involving the variables of this cycle can, thus, be simplified to:

$$\left[\bigwedge_{j \in J - \{e\}} (x_j = x_j^\pi) \right] = (x_s = x_e)$$

and since $(x_s = x_e) \rightarrow (x_e \leq x_s)$ is a tautology, the whole bit predicate becomes tautologous. Elimination of these BPs is accomplished by a further restriction of the index set in (13.24) and (13.25) to just $\text{supp}(\pi) - \text{ends}(\pi)$ and corresponds to a reduction in the number of BPs from n to $m = |\text{supp}(\pi)| - \text{cycles}(\pi)$ where $\text{cycles}(\pi)$ is the number of non-phase-shift cycles of π .

The third and last case corresponds to the BPs of those bits that occur after the first “phase-shifted variable.” Let i be the index of the first variable for which $x_i^\pi = x_i'$. Thus, $e_i = 0$ and all BPs for $j < i$ have the form $0 \rightarrow (x_j \leq x_j^\pi)$ making them unconditionally true.

⁷Technically, we should replace I_n with $\{i \in I_n | \dot{x}_i \in \text{supp}(\pi)\}$.

$$\pi = \begin{pmatrix} x_9 & x_8 & x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 \\ x_6 & x_8 & x_2 & x_9 & x'_5 & x'_7 & x_3 & x'_4 & x_1 & x_0 \end{pmatrix} = (x_9, x_6)(x_7, x_2, x'_4)(x_5, x'_5)$$

$$\text{supp}(\pi) = \{2, 4, 5, 6, 7, 9\}, \text{phase-shift}(\pi) = 5, \text{ends}(\pi) = \{2, 6\}$$

$$I = \text{supp}(\pi) - \text{ends}(\pi) - \text{pred}(\text{phase-shift}(\pi), I_{10}) = \{5, 7, 9\}$$

(a) Permutation in tabular and cycle notation, along with various associated index sets.

$$\text{BP}(\pi, x_9) = (x_9 \leq x_6)$$

$$\boxed{\text{BP}(\pi, x_8) = (x_9 = x_6) \rightarrow (x_8 \leq x_8)}$$

$$\text{BP}(\pi, x_7) = (x_9 = x_6)(x_8 = x_8) \rightarrow (x_7 \leq x_2)$$

$$\boxed{\text{BP}(\pi, x_6) = (x_9 = x_6)(x_8 = x_8)(x_7 = x_2) \rightarrow (x_6 \leq x_9)}$$

$$\text{BP}(\pi, x_5) = (x_9 = x_6)(x_8 = x_8)(x_7 = x_2)(x_6 = x_9) \rightarrow (x_5 \leq x'_5)$$

$$\boxed{\text{BP}(\pi, x_4) = (x_9 = x_6)(x_8 = x_8)(x_7 = x_2)(x_6 = x_9)(x_5 = x'_5) \rightarrow (x_4 \leq x'_7)}$$

$$\boxed{\text{BP}(\pi, x_3) = (x_9 = x_6)(x_8 = x_8)(x_7 = x_2)(x_6 = x_9)(x_5 = x'_5)(x_4 = x'_7) \rightarrow (x_3 \leq x_3)}$$

$$\boxed{\text{BP}(\pi, x_2) = (x_9 = x_6)(x_8 = x_8)(x_7 = x_2)(x_6 = x_9)(x_5 = x'_5)(x_4 = x'_7)(x_3 = x_3) \rightarrow (x_2 \leq x'_4)}$$

$$\boxed{\text{BP}(\pi, x_1) = (x_9 = x_6)(x_8 = x_8)(x_7 = x_2)(x_6 = x_9)(x_5 = x'_5)(x_4 = x'_7)(x_3 = x_3)(x_2 = x'_4) \rightarrow (x_1 \leq x_1)}$$

$$\boxed{\text{BP}(\pi, x_0) = (x_9 = x_6)(x_8 = x_8)(x_7 = x_2)(x_6 = x_9)(x_5 = x'_5)(x_4 = x'_7)(x_3 = x_3)(x_2 = x'_4)(x_1 = x_1) \rightarrow (x_0 \leq x_0)}$$

(b) Permutation's bit predicates. BPs enclosed in boxes with square corners are tautologous because π maps the corresponding bits to themselves. BPs enclosed in boxes with rounded corners are tautologous because they correspond to cycle "ends." The BPs for bits 4 to 0 are tautologous because π maps bit 5 to its complement.

$$\begin{aligned} \text{PP}^*(\pi; X) &= (l_9)(p_9 \rightarrow l_7)(p_7 \rightarrow l_5)(g_9 \rightarrow p_9)(p_9 g_7 \rightarrow p_7) \\ &= (x'_9 + x_6)(p'_9 + x'_7 + x_2)(p'_7 + x'_5)(x'_9 + p_9)(x_6 + p_9)(p'_9 + x'_7 + p_7)(p'_9 + x_2 + p_7) \end{aligned}$$

(c) Linear formulation of the (relaxed) permutation predicate using the chaining 'p' variables.

Figure 13.9. Illustration of the formulation of the permutation predicate according to (13.26) and (13.32).

Taken together, the redundant BPs corresponding to these three cases can be easily eliminated by setting the index set in (13.24) and (13.25) to:

$$I = \text{supp}(\pi) - \text{ends}(\pi) - \text{pred}(\text{phase-shift}(\pi), I_n) \quad (13.26)$$

In the sequel we will refer to the bits in the above index set as “irredundant bits” and use $I = \{i_1, i_2, \dots, i_m\}$, with $i_1 > i_2 > \dots > i_m$, to label them. Note that the presence of a phase-shifted variable early in the total order can lead to a drastic reduction in the number of irredundant bits. For example, if $\pi = (x_{n-1}, x'_{n-1}) \dots$ then $\text{PP}(\pi; X)$ is simply (x'_{n-1}) regardless of how many other variables are moved by π .

Linear construction of PPs through chaining. Elimination of the redundant bit predicates yields the following reduced expression for $\text{PP}(\pi; X)$:

$$\text{PP}(\pi; X) = \bigwedge_{1 \leq k \leq m} \left\{ \left[\bigwedge_{1 \leq j \leq k-1} (x_{i_j}^\pi = x_{i_j}) \right] \rightarrow (x_{i_k}^\pi \leq x_{i_k}) \right\} \quad (13.27)$$

where the outer conjunction is now over the irredundant index set $\{i_1, \dots, i_m\}$. We show next how to convert (13.27) to a CNF formula whose size is linear in m . The first step in this conversion is to eliminate the equalities in the inner conjunction. Specifically, we introduce the “ordering” predicates $l_{i_j} = (x_{i_j} \leq x_{i_j}^\pi)$ and $g_{i_j} = (x_{i_j} \geq x_{i_j}^\pi)$ to re-write (13.27) as:

$$\text{PP}(\pi; X) = \bigwedge_{1 \leq k \leq m} \left\{ \left[\bigwedge_{1 \leq j \leq k-1} l_{i_j} g_{i_j} \right] \rightarrow l_{i_k} \right\} \quad (13.28)$$

The next step utilizes the following easily-proved lemma:

Lemma 13.7.1. (Elimination of Redundant Premise)

$$(a \rightarrow b) \wedge \bigwedge_{i \in I} (abc_i \rightarrow d_i) = (a \rightarrow b) \wedge \bigwedge_{i \in I} (ac_i \rightarrow d_i)$$

Repeated application of this lemma to (13.28) leads to the successive elimination of the “less-than-or-equal” predicates in the inner conjunction:

$$\begin{aligned} & \text{PP}(\pi; X) \\ &= (1 \rightarrow l_{i_1}) (l_{i_1} g_{i_1} \rightarrow l_{i_2}) (l_{i_1} g_{i_1} l_{i_2} g_{i_2} \rightarrow l_{i_3}) \dots (l_{i_1} g_{i_1} \dots l_{i_{m-1}} g_{i_{m-1}} \rightarrow l_{i_m}) \\ &= (1 \rightarrow l_{i_1}) (g_{i_1} \rightarrow l_{i_2}) (g_{i_1} l_{i_2} g_{i_2} \rightarrow l_{i_3}) \dots (g_{i_1} l_{i_2} g_{i_2} \dots l_{i_{m-1}} g_{i_{m-1}} \rightarrow l_{i_m}) \\ &= (1 \rightarrow l_{i_1}) (g_{i_1} \rightarrow l_{i_2}) (g_{i_1} g_{i_2} \rightarrow l_{i_3}) \dots (g_{i_1} g_{i_2} \dots l_{i_{m-1}} g_{i_{m-1}} \rightarrow l_{i_m}) \quad (13.29) \\ &= \dots \\ &= (1 \rightarrow l_{i_1}) (g_{i_1} \rightarrow l_{i_2}) (g_{i_1} g_{i_2} \rightarrow l_{i_3}) \dots (g_{i_1} g_{i_2} \dots g_{i_{m-1}} \rightarrow l_{i_m}) \\ &= \bigwedge_{1 \leq k \leq m} \left\{ \left[\bigwedge_{1 \leq j \leq k-1} g_{i_j} \right] \rightarrow l_{i_k} \right\} \end{aligned}$$

Next, we decompose the multi-way conjunctions involving the “greater-than-or-equal” predicates by introducing m auxiliary *chaining* variables⁸ defined by:

$$\begin{aligned}
 p_{i_0} &= 1 \\
 p_{i_1} &= p_{i_0} \wedge g_{i_1} = g_{i_1} \\
 p_{i_2} &= p_{i_1} \wedge g_{i_2} = g_{i_1} \wedge g_{i_2} \\
 &\dots \\
 p_{i_{m-1}} &= p_{i_{m-2}} \wedge g_{i_{m-1}} = \bigwedge_{1 \leq j \leq m-1} g_{i_j}
 \end{aligned} \tag{13.30}$$

Substituting these definitions, along with those of the ordering predicates, yields the following final expression for the permutation predicate

$$\text{PP}(\pi; X) = \left[\bigwedge_{1 \leq k \leq m} (p_{i_{k-1}} \rightarrow (x_{i_k} \leq x_{i_k}^\pi)) \right] \wedge \left[\bigwedge_{1 \leq k \leq m-1} (p_{i_k} = p_{i_{k-1}} \wedge (x_{i_k} \geq x_{i_k}^\pi)) \right] \tag{13.31}$$

which can be readily converted to a CNF formula consisting of, approximately, $4m$ 3-literal clauses and m 2-literal clauses for a total of $14m$ literals. A further reduction is possible by replacing the equalities in the second conjunction in (13.31) with one-way implications, effectively relaxing the permutation predicate to

$$\text{PP}^*(\pi; X) = \left[\bigwedge_{1 \leq k \leq m} (p_{i_{k-1}} \rightarrow (x_{i_k} \leq x_{i_k}^\pi)) \right] \wedge \left[\bigwedge_{1 \leq k \leq m-1} (p_{i_{k-1}} \wedge (x_{i_k} \geq x_{i_k}^\pi) \rightarrow p_{i_k}) \right] \tag{13.32}$$

Since $\text{PP}(\pi; X) \leq \text{PP}^*(\pi; X)$, it is now possible for $\text{PP}^*(\pi; X)$ to have solutions that violate $\text{PP}(\pi; X)$. These solutions follow the pattern:

$$\begin{array}{ccc}
 p_{i_{k-1}} & g_{i_k} & p_{i_k} \\
 0 & 0 & 1 \\
 0 & 1 & 1 \\
 1 & 0 & 1
 \end{array}$$

In other words, p_{i_k} is 1 when either $p_{i_{k-1}}$ or g_{i_k} is 0. Each such solution, however, can be modified to a solution that does satisfy $\text{PP}(\pi; X)$ by simply changing the value of p_{i_k} from 1 to 0. We can thus use $\text{PP}^*(\pi; X)$ without compromising correctness; this predicate consists of $3m$ 3-literal clauses for a total of $9m$ literals.

13.7.2. Partial Symmetry Breaking

By filtering all but the least assignment in each orbit of the symmetry group G_φ , the SBP in (13.16) is able to break every symmetry of the associated CNF formula

⁸Actually, we only need $m - 1$ auxiliary variables since p_{i_0} stands for the constant 1.

$\varphi(X)$. For exponentially large symmetry groups, however, such *full symmetry breaking* is infeasible. Even when it is possible to construct a full SBP, its ability to prune away symmetric assignments in the search space will be overwhelmed by a drastic increase in search time caused by the sheer size of the formula that must be processed by the SAT solver. The only option left in such situations is to abandon the goal of full symmetry breaking and to re-phrase the objective of the exercise as that of minimizing SAT search time by breaking some, but not necessarily all, symmetries. Like a full SBP, a *partial* SBP selects the least assignment in each orbit of the symmetry group but may include other assignments as well. Formally, if \hat{G}_φ is a subset of G_φ then

$$\rho(G_\varphi; X) \leq \rho(\hat{G}_\varphi; X) \tag{13.33}$$

A natural choice for \hat{G}_φ is any set of irredundant generators for the group. Such a set is computationally attractive because, by Theorem 13.3.3, it is guaranteed to be exponentially smaller than the order of the symmetry group. Additionally, the independence of the generators in an irredundant set (i.e., the fact that none of the generators can be expressed in terms of any others) suggests that their corresponding SBP may block more symmetric assignments than that, say, of an equally-sized set of randomly-chosen elements of G_φ .

Table 13.11 lists the nine sets of irredundant generators for the symmetry group of the example function from Table 13.9 along with their corresponding SBPs. Three of these generator sets yield full SBPs, whereas the remaining six produce partial SBPs. Interestingly, the set $\{\pi_4, \pi_5\}$ which does not generate the group also yields a full SBP! Thus, choosing to break the symmetries of a set of irredundant generators must be viewed as a heuristic whose effectiveness can only be judged empirically.

Table 13.11. Symmetry-breaking predicates of different generator sets for the symmetry group of the CNF formula in Table 13.9. Note that the SBPs of three generator sets achieves full symmetry breaking, whereas those of the remaining six achieve only partial symmetry breaking.

Generator Set	Lex-Leader Predicate	Symmetry Breaking
$\{\pi_1, \pi_2\}$	$\sum_{(a,b,c)} (0, 1, 2, 3, 6) = (a' + b)(a' + c')$	Partial
$\{\pi_1, \pi_3\}$	$\sum_{(a,b,c)} (0, 1, 2, 6) = (a' + b)(b' + c')$	Full
$\{\pi_1, \pi_4\}$	$\sum_{(a,b,c)} (0, 1, 2, 3, 6) = (a' + b)(a' + c')$	Partial
$\{\pi_1, \pi_5\}$	$\sum_{(a,b,c)} (0, 1, 2, 6) = (a' + b)(b' + c')$	Full
$\{\pi_2, \pi_3\}$	$\sum_{(a,b,c)} (0, 1, 2, 4, 6) = (a' + c')(b' + c')$	Partial
$\{\pi_2, \pi_4\}$	$\sum_{(a,b,c)} (0, 1, 2, 3, 6) = (a' + b)(a' + c')$	Partial
$\{\pi_2, \pi_5\}$	$\sum_{(a,b,c)} (0, 1, 2, 4, 6) = (a' + c')(b' + c')$	Partial
$\{\pi_3, \pi_4\}$	$\sum_{(a,b,c)} (0, 1, 2, 6) = (a' + b)(b' + c')$	Full
$\{\pi_3, \pi_5\}$	$\sum_{(a,b,c)} (0, 1, 2, 4, 6) = (a' + c')(b' + c')$	Partial

13.7.3. Experimental Evaluation of Symmetry Breaking

The impact of symmetry breaking on SAT search time for the benchmark instances in Table 13.8 is shown in Table 13.12. For each instance, the table lists the following data:

(1, -1) (5, -5) (12, -12) (13, -13) (19, -19)
 (13, -13) (22, -22) (31, -31) (39, -39) (45, -45)
 (2, -2) (5, -5) (13, -13) (19, -19) (22, -22)
 (5, -5) (9, -9) (13, -13) (22, -22) (28, -28) (31, -31) (39, -39) (41, -41)
 (1, -1) (11, -11) (22, -22) (31, -31) (32, -32) (39, -39) (41, -41) (44, -44)
 (11, -11) (25, -25) (35, -35)
 (10, -10) (11, -11) (13, -13) (22, -22) (29, -29) (31, -31) (39, -39) (41, -41)
 (1, -1) (11, -11) (22, -22) (30, -30) (31, -31) (38, -38) (39, -39) (41, -41)
 (16, -16) (22, -22) (23, -23) (37, -37)
 (16, -16) (20, -20) (26, -26) (31, -31) (39, -39) (41, -41)
 (16, -16) (17, -17) (25, -25) (31, -31) (39, -39) (41, -41)
 (1, -1) (6, -6) (16, -16) (22, -22) (38, -38)
 (5, -5) (13, -13) (23, -23) (28, -28) (42, -42)
 (10, -10) (13, -13) (23, -23) (34, -34)
 (1, -1) (21, -21) (23, -23) (38, -38)
 (4, -4) (26, -26) (41, -41)
 (3, -3) (25, -25) (41, -41)
 (5, -5) (13, -13) (22, -22) (26, -26) (28, -28) (31, -31) (39, -39) (40, -40) (41, -41)
 (8, -8) (25, -25) (26, -26)
 (1, -1) (22, -22) (26, -26) (31, -31) (38, -38) (39, -39) (41, -41) (46, -46)
 (1, -1) (5, -5) (13, -13) (28, -28) (36, -36) (44, -44)
 (1, -1) (10, -10) (13, -13) (18, -18) (44, -44)
 (15, -15) (38, -38) (44, -44)
 (1, -1) (5, -5) (13, -13) (19, -19) (33, -33) (44, -44)
 (14, -14) (19, -19) (28, -28)
 (10, -10) (13, -13) (22, -22) (25, -25) (31, -31) (39, -39) (41, -41) (43, -43)
 (5, -5) (7, -7) (13, -13) (19, -19) (22, -22) (25, -25) (31, -31) (39, -39) (41, -41)
 (1, -1) (10, -10) (13, -13) (24, -24) (38, -38)
 (5, -5) (10, -10) (19, -19) (27, -27)

(a) Generators for Mixed Symmetries with corresponding SBP $\bigwedge_{i \in S} x'_i$ where
 $S = \{1-5, 8, 10, 11, 13-16\}$

(36, -36) (38, -38) (40, -40) (44, -44) (46, -46)
 (32, -32) (34, -34) (35, -35) (38, -38) (40, -40) (42, -42) (43, -43) (44, -44) (46, -46)
 (30, -30) (34, -34) (35, -35) (40, -40) (42, -42) (43, -43) (46, -46)
 (29, -29) (35, -35) (43, -43)
 (27, -27) (33, -33) (34, -34) (38, -38) (40, -40) (42, -42) (44, -44) (46, -46)
 (25, -25) (26, -26) (34, -34) (40, -40) (42, -42) (43, -43)
 (24, -24) (34, -34) (40, -40) (42, -42) (46, -46)
 (22, -22) (23, -23) (26, -26) (31, -31) (39, -39) (40, -40) (41, -41) (42, -41)
 (21, -21) (40, -40) (42, -42) (46, -46)
 (20, -20) (37, -37) (40, -40) (42, -41)
 (19, -19) (28, -28) (33, -33) (38, -38) (40, -40) (44, -44) (46, -46)
 (18, -18) (34, -34) (38, -38) (40, -40) (42, -42) (44, -44) (46, -46)
 (17, -17) (34, -34) (37, -37) (43, -43)
 (16, -16) (26, -26) (31, -31) (37, -37) (39, -39) (40, -40) (41, -41) (42, -42)
 (15, -15) (38, -38) (44, -44)
 (14, -14) (33, -33) (38, -38) (40, -40) (44, -44) (46, -46)
 (13, -13) (23, -23) (26, -26) (40, -40) (41, -41) (42, -42) (45, -45)
 (12, -12) (33, -33) (44, -44)
 (11, -11) (26, -26) (34, -34) (35, -35) (40, -40) (42, -42) (43, -43)
 (10, -10) (26, -26) (34, -34) (40, -40) (41, -41) (42, -42) (45, -45)
 (9, -9) (26, -26) (40, -40)
 (8, -8) (34, -34) (40, -40) (42, -42) (43, -43)
 (7, -7) (33, -33) (34, -34) (38, -38) (40, -40) (42, -42) (43, -43) (44, -44) (46, -46)
 (6, -6) (37, -37) (40, -40) (42, -42) (46, -46)
 (5, -5) (26, -26) (28, -28) (40, -40) (41, -41) (45, -45)
 (4, -4) (26, -26) (41, -41)
 (2, -2) (26, -26) (31, -31) (33, -33) (38, -38) (39, -39) (41, -41) (44, -44) (46, -46)
 (3, -3) (26, -26) (34, -34) (40, -40) (41, -41) (42, -42) (43, -43)
 (1, -1) (23, -23) (38, -38) (40, -40) (42, -42) (46, -46)

(b) Generators for Value Symmetries with corresponding SBP $\bigwedge_{i \in S} x'_i$ where
 $S = \{1-22, 24, 25, 27, 29, 30, 32, 36\}$

Figure 13.10. Generators for the Urq3.5 benchmark

- Instance parameters: family, name, number of variables, number of clauses, and number of literals
- SAT search time without symmetry breaking
- SBP size for each variant (mixed, variable, and value): number of variables, number of clauses, and number of literals
- SAT search time when the instance is conjoined with each of the three SBP variants

The SBPs in these experiments were created using the optimization described in [ASM03] which yields a permutation predicate consisting of $4|I|$ clauses and $14|I|$ literals where I is the index of irredundant bits specified in (13.26). SAT checking was carried out using the RSAT solver [RSA] which won the gold medal in the industrial category in the SAT'07 competition [SAT]. The experiments were performed on a 3GHz Intel Xeon workstation with 4GB RAM running the Redhat Linux Enterprise operating system. Each instance was allowed to run for up to 1000 seconds; times-outs are indicated in the table by '-'.²

Close examination of these results leads to the following general observations:

- On average, the size of the SBP-augmented formula (number of variables and number of literals) is slightly more than twice the size of the original formula. The largest increase in size (about 3.4X for variables and 4X for literals) occurred for the Hole, FPGARoute, and ChnlRoute benchmarks; the smallest (about 1.3X for variables and 1.1X for literals) was for the Urq and Pipe benchmarks. These are relatively modest increases that should not tax the abilities of modern SAT solvers.
- Generally, but not always, the addition of the SBP helps to reduce the time to perform the SAT check. Specifically,
 - The Hole, ChnlRoute, Urq, and XOR instances all benefited from the addition of the SBP, even when the symmetry detection time was factored in. This was especially true for instances which could not be solved within the time-out limit.
 - The FPGARoute instances were quite easy to solve without the addition of the SBPs. There was a marginal speed-up in search time when the SBPs were added. However, when symmetry detection times were factored in, the net effect was a marked slow-down.
 - The results for the GRoute instances were mixed. The addition of SBPs improved search times in some cases but worsened it in others. When symmetry detection times were added in, however, the net effect was a noticeable slow-down.
 - The Pipe instances all suffered increases in search times when SBPs were added, even without considering symmetry detection times.
- In cases where only pure variable symmetries are present (Hole, ChnlRoute, FPGARoute, and GRoute), comparable search times were obtained for the two variants of an augmented instance (namely the instance augmented with the variable symmetry SBP and the one augmented with the mixed symmetry SBP).
- However, in cases where only pure value symmetries are present (Urq and all but the smallest XOR), the search times for instances augmented with

Table 13.12. SAT search times (in seconds) with and without symmetry breaking.

Benchmark		Symmetry Breaking Predicate																
Family	Name	Parameters				Time	Mixed Symmetries				Variable Symmetries				Value Symmetries			
		Vars	Clauses	Lits			Vars	Clauses	Lits	Time	Vars	Clauses	Lits	Time	Vars	Clauses	Lits	Time
Hole	hole7	56	204	448	0.03	97	388	1358	0	97	388	1358	0	0	0	0	0.03	
	hole8	72	297	648	0.18	127	508	1778	0	127	508	1778	0	0	0	0	0.18	
	hole9	90	415	900	50.19	161	644	2254	0.01	161	644	2254	0	0	0	0	50.19	
	hole10	110	561	1210	336.86	199	796	2786	0	199	796	2786	0.01	0	0	0	336.86	
	hole11	132	738	1584	-	241	964	3374	0	241	964	3374	0	0	0	0	-	
	hole12	156	949	2028	-	287	1148	4018	0	287	1148	4018	0	0	0	0	-	
ChnlRoute	chnl10_11	220	1122	2420	-	508	2032	7112	0	508	2032	7112	0.01	0	0	0	-	
	chnl10_12	240	1344	2880	-	556	2224	7784	0	556	2224	7784	0.01	0	0	0	-	
	chnl10_13	260	1586	3380	-	604	2416	8456	0	604	2416	8456	0.01	0	0	0	-	
	chnl11_12	264	1476	3168	-	614	2456	8596	0	614	2456	8596	0	0	0	0	-	
	chnl11_13	286	1742	3718	-	667	2668	9338	0.01	667	2668	9338	0	0	0	0	-	
	chnl11_20	440	4220	8800	-	1038	4152	14532	0	1038	4152	14532	0.01	0	0	0	-	
FPGAARoute	fpga10_8	120	448	1200	0.02	256	1024	3584	0	256	1024	3584	0	0	0	0	0.02	
	fpga10_9	135	549	1485	0.01	223	892	3122	0	223	892	3122	0	0	0	0	0.01	
	fpga12_8	144	560	1488	0.01	312	1248	4368	0	312	1248	4368	0	0	0	0	0.01	
	fpga12_9	162	684	1836	0.04	273	1092	3822	0	273	1092	3822	0.01	0	0	0	0.04	
	fpga12_11	198	968	2640	0.02	339	1356	4746	0.01	339	1356	4746	0	0	0	0	0.02	
	fpga12_12	216	1128	3096	0.01	480	1920	6720	0.01	480	1920	6720	0	0	0	0	0.01	
	fpga13_9	176	759	2031	0.01	299	1196	4186	0.01	299	1196	4186	0.01	0	0	0	0.01	
	fpga13_10	195	905	2440	0.04	334	1336	4676	0.01	334	1336	4676	0	0	0	0	0.04	
	fpga13_12	234	1242	3396	0.01	406	1624	5684	0	406	1624	5684	0.01	0	0	0	0.01	
	Groute	s3-3-3-1	864	7592	16038	0.08	528	2112	7392	0.74	528	2112	7392	0.12	0	0	0	0.08
s3-3-3-3		960	9156	19258	0.06	720	2880	10080	0.16	720	2880	10080	0.16	0	0	0	0.06	
s3-3-3-4		912	8356	17612	0.82	600	2400	8400	0.34	600	2400	8400	0.09	0	0	0	0.82	
s3-3-3-8		912	8356	17612	0.41	648	2592	9072	0.27	648	2592	9072	1.25	0	0	0	0.41	
s3-3-3-10		1056	10862	22742	2.36	720	2880	10080	0.09	720	2880	10080	0.08	0	0	0	2.36	
Urq	Urq3_5	46	470	2912	354.34	29	116	406	0.1	0	0	0	354.34	29	116	406	0	
	Urq4_5	74	694	4272	-	43	172	602	122.73	0	0	0	-	43	172	602	0	
	Urq5_5	121	1210	7548	-	72	288	1008	-	0	0	0	-	72	288	1008	0	
	Urq6_5	180	1756	10776	-	109	436	1526	-	0	0	0	-	109	436	1526	0	
	Urq7_5	240	2194	13196	-	143	572	2002	-	0	0	0	-	143	572	2002	0.01	
	Urq8_5	327	3252	20004	-	200	800	2800	-	0	0	0	-	200	800	2800	0	
XOR	x1_16	46	122	364	0.02	18	72	252	0	1	4	14	0.02	16	64	224	0.01	
	x1_24	70	186	556	34.5	24	96	336	0.01	0	0	0	34.5	24	96	336	0	
	x1_32	94	250	748	1.33	32	128	448	0	0	0	0	1.33	32	128	448	0	
	x1_36	106	282	844	181.51	36	144	504	0.29	0	0	0	181.51	36	144	504	0	
Pipe	2pipe_1_ooo	834	7026	19768	0.11	321	1284	4494	0.25	317	1268	4438	0.14	0	0	0	0.11	
	2pipe_2_ooo	925	8213	23161	0.12	362	1448	5068	0.14	354	1416	4956	0.19	0	0	0	0.12	
	2pipe	861	6695	18637	0.13	358	1432	5012	0.16	350	1400	4900	0.21	0	0	0	0.13	

mixed symmetry SBPs were worse (some timing out) than the search times for the same instances augmented with value symmetry SBPs. The only plausible explanation of such divergent behaviors is that the mixed and value symmetry SBPs are derived from two different sets of generators. This turned out to be the case as illustrated in Fig. 13.10. The figure shows the generators returned by **saucy** for the 46-variable Urq3.5 benchmark: part (a) shows the generators obtained for mixed symmetries and part (b) gives the generators obtained for value symmetries. In both cases, the generators consist of phase-shift permutations (as expected), but the two sets are different. This leads to different SBPs which, in this case, are simply conjunctions of negative literals that correspond to the phase-shift cycle of the variable that occurs first in the assumed variable ordering. Clearly, the generators obtained for value symmetries lead to a SBP that is superior to that obtained from the generators for mixed symmetries. Specifically, the SBP for value symmetries “fixes” 29 out of the 46 variables compared to the SBP for mixed symmetries which fixes only 12 variables. This difference in pruning power was not sufficient to cause the search times to be substantially different for this particular instance. For the larger instances, however, the pruning ability of the SBPs derived from value symmetries is clearly evident; in fact the SBPs derived from mixed symmetries were unsuccessful in helping the SAT solver prove the unsatisfiability of their corresponding instances within the 1000 second time-out.

13.8. From Symmetry to Satisfiability and Back⁹

While the intuition behind exploiting symmetry to speed up SAT search is compelling, as of this writing the empirical evidence is still inconclusive and this quest remains an active research area. On the other hand, and in an unexpected twist, this line of inquiry led to a re-examination of established algorithms for symmetry detection and to their re-formulation as *search* for graph automorphisms in the *space of vertex permutations* yielding orders-of-magnitude improvements in capacity and performance over traditional approaches.

The initial trigger was the black-box use of the **nauty** graph automorphism and canonical labeling package [McK81, McK] to detect the symmetries in CNF formulas as described in Section 13.6. It quickly became apparent, however, that the graphs of typical CNF formulas were too large (hundreds of thousands to millions of vertices) and unwieldy for **nauty** which was geared more towards small dense graphs (hundreds of vertices). The obvious remedy, changing the data structure for storing graphs from an incidence matrix to a linked list, yielded the **saucy** system which demonstrated the viability of graph automorphism detection on very large sparse graphs [DLSM04]. Unlike **nauty**, which also solved the canonical labeling problem, **saucy** was limited to just finding an irredundant set of symmetry generators. Canonical labeling seeks to assign a unique signature to a graph that captures its structure and is invariant under all possible labelings of its vertices. The **bliss** tool [JK07] adopted, and improved upon, **saucy**'s sparse data

⁹Material in this section is based on [KSM10].

structures and solved both the symmetry detection and canonical labeling problems for small dense and large sparse graphs. Close analysis of the search trees used in **nauty** and **bliss** revealed that they were primarily designed to solve the canonical labeling problem, and that symmetry generators were detected “along the way.” Both tools employed sophisticated group-theoretic pruning heuristics to narrow the search for the canonical labeling of an input graph. The detection of symmetries benefited from these pruning rules, but discovered symmetries also helped prune the “canonical labeling” tree since labelings that are related by a symmetry yield the same signature.

Recognizing this, the next version of **saucy** [DSM08] replaced the canonical labeling tree (see Fig. 13.7) with a *permutation search tree* in order to process vertex permutations directly without being constrained by the need to also solve the canonical labeling problem. Instead of a tree of single ordered partitions, the nodes in the **saucy** tree are *ordered partition pairs* that represent partial permutations in analogy with partial Boolean assignments in a SAT search tree. Partial permutations allow the encoding of *sets* of permutations and enable significant additional pruning of the search tree beyond what’s possible with group-theoretic pruning.

De-linking the search for symmetries from the search for a canonical labeling was a major algorithmic change that yielded a remarkable 1000-fold improvement in run time for many large sparse graphs with sparse symmetry generators, i.e., generators that “move” only a tiny fraction of the graph’s vertices. This change also made the search for symmetries resemble, at least superficially, the search for satisfying assignments by a SAT solver.

13.8.1. Implicit Representation of Permutation Sets

Searching the space of a graph’s vertex permutations is greatly facilitated by the adoption of a data structure that compactly represents not just single but also sets of permutations.

Definition 13.8.1. (Ordered Partition Pair or OPP) An *ordered partition pair* Π is specified as

$$\Pi = \begin{bmatrix} \pi_T \\ \pi_B \end{bmatrix} = \begin{bmatrix} T_1 | T_2 | \cdots | T_m \\ B_1 | B_2 | \cdots | B_k \end{bmatrix} \quad (13.34)$$

with π_T and π_B referred to, respectively, as its top and bottom ordered partitions. An OPP is *isomorphic* if $m = k$ and $|T_i| = |B_i|$ for $i = 1, \dots, m$; otherwise it is *non-isomorphic*. In other words, it is isomorphic if its top and bottom partitions have the same number of cells, and corresponding cells have the same cardinality. An isomorphic OPP

$$\Pi = \begin{bmatrix} T_1 & T_2 & \cdots & T_m \\ B_1 & B_2 & \cdots & B_m \end{bmatrix} \quad (13.35)$$

can be further classified, based on the structure and content of its corresponding cells, as:

$$\text{Unmapped: } \bigwedge_{1 \leq i \leq m} (T_i = B_i) \quad (13.36)$$

$$\text{Matching: } \bigvee_{i \in \text{Singletons}(\Pi)} (T_i \neq B_i) \wedge \bigwedge_{i \notin \text{Singletons}(\Pi)} (T_i = B_i) \quad (13.37)$$

$$\text{Non-Matching: } \bigvee_{i \in \text{Singletons}(\Pi)} (T_i \neq B_i) \wedge \bigvee_{i \notin \text{Singletons}(\Pi)} (T_i \neq B_i) \quad (13.38)$$

where $\text{Singletons}(\Pi) = \{i \mid |T_i| = 1\}$. Finally an OPP is discrete (resp. unit) if its top and bottom partitions are discrete (resp. unit).

An isomorphic OPP is an implicit compact representation of a set of $\prod_{1 \leq i \leq m} |T_i|!$ permutations. Thus, a unit OPP represents all $n!$ permutations of V , and a discrete OPP represents just one permutation. On the other hand, it is not possible to obtain well-defined mappings between the top and bottom partitions of a non-isomorphic OPP. Thus, non-isomorphic OPPs conveniently serve as empty sets of permutations.

An OPP can be viewed as a generalization of the tabular format for representing single permutations. Specifically, the set of permutations corresponding to an unmapped OPP can be expressed as

$$\Pi = \bigcirc_{1 \leq i \leq m} S_{|T_i|}(T_i) \quad (13.39)$$

where function composition \bigcirc is naturally extended from single permutations to sets of permutations. The set of permutations corresponding to a matching OPP can also be expressed succinctly as

$$\Pi = \gamma(\text{Singletons}(\Pi)) \circ \bigcirc_{i \notin \text{Singletons}(\Pi)} S_{|T_i|}(T_i) \quad (13.40)$$

where $\gamma(\text{Singletons}(\Pi))$ is the permutation formed from the singleton cells of Π . Fig. 13.11 shows several different examples of OPPs and the permutation sets they represent.

13.8.2. Permutation Search Tree

OPP's play a role similar to partial variable assignments in CNF-SAT solvers. Recall that a partial variable assignment on n Boolean variables can be encoded by an n -element array whose i th element indicates the value of the i th variable: 0, 1, or X for *unassigned*. A *complete* assignment is one in which all variables have been assigned a binary value; otherwise the assignment is *partial* and corresponds to a set of complete assignments that can be enumerated by considering all possible 0, 1 combinations of the unassigned variables. A backtracking SAT solver extends a given partial assignment by choosing an unassigned variable and assigning to it one of the two binary values. This is referred to as a *decision* step and SAT solvers use a variety of decision heuristics to determine which variable to assign next and what value to assign to it. SAT solvers also employ *propagation* to avoid making decisions on variables whose values are implied (forced) by prior decisions. Finally, SAT solvers backtrack from conflicts, i.e. assignments that cause the formula being checked to become unsatisfied.

$$\begin{aligned}
 \text{(a)} \quad & \left[\begin{array}{c|c|c|c} 3 & 1 & 2 & 4 \\ 2 & 3 & 1 & 4 \end{array} \right] = \{(1\ 3\ 2)\} \\
 \text{(b)} \quad & \left[\begin{array}{c|c} 1, 2, 3 \\ 1, 2, 3 \end{array} \right] = S_3(\{1, 2, 3\}) = \{\iota, (1\ 2), (1\ 3), (2\ 3), (1\ 2\ 3), (1\ 3\ 2)\} \\
 \text{(c)} \quad & \left[\begin{array}{c|c|c} 1, 3 & 2, 5 & 4 \\ 1, 3 & 2, 5 & 4 \end{array} \right] = S_2(\{1, 3\}) \circ S_2(\{2, 5\}) = \{\iota, (1\ 3), (2\ 5), (1\ 3)(2\ 5)\} \\
 \text{(d)} \quad & \left[\begin{array}{c|c|c|c} 2 & 6 & 1, 3, 5 & 4 \\ 4 & 6 & 1, 3, 5 & 2 \end{array} \right] = (2\ 4) \circ S_3(\{1, 3, 5\}) \\
 \text{(e)} \quad & \left[\begin{array}{c|c|c} 4 & 1, 2 & 3 \\ 2 & 4, 1 & 3 \end{array} \right] = \{(2\ 4), (1\ 4\ 2)\} \\
 \text{(f)} \quad & \left[\begin{array}{c|c} 1, 3 | 2 \\ 2 | 3, 1 \end{array} \right] = \left[\begin{array}{c|c} 3 | 1 | 2 \\ 2 | 3, 1 \end{array} \right] = \emptyset
 \end{aligned}$$

Figure 13.11. Example OPPs and the permutation sets they represent: (a) discrete, (b) unit, (c) unmapped, (d) matching, (e) non-matching, (f) non-isomorphic. The OPPs (a)-(e) are all isomorphic.

The basic skeleton of a permutation enumeration algorithm can thus be patterned after a backtracking SAT algorithm that finds *all* satisfying assignments to a given CNF formula. The decision step in such an algorithm starts with an isomorphic OPP and refines it by:

- choosing a non-singleton cell (the *target* cell) from the top partition
- choosing a vertex from the target cell (the *target* vertex), and
- mapping the target vertex to a vertex (the *mapped* vertex) from the corresponding cell (the *mapped* cell) of the bottom partition.

The mapping step is accomplished by simultaneously splitting the target vertex from the target cell and the mapped vertex from the mapped cell and placing them in their own singleton cell. Symbolically, given the isomorphic OPP in (13.35) assume that the i th cell is the target cell and let $j \in T_i$ be the target vertex. Mapping j to $k \in B_i$ refines the m -cell OPP Π to the following $(m+1)$ -cell OPP Π' :

$$\Pi' = \left[\begin{array}{c|c|c|c|c|c|c} T'_1 & T'_2 & \cdots & T'_i & T'_{i+1} & \cdots & T'_{m+1} \\ B'_1 & B'_2 & \cdots & B'_i & B'_{i+1} & \cdots & B'_{m+1} \end{array} \right]$$

where

$$\begin{aligned}
 T'_l &= T_l & B'_l &= B_l & l &= 1, \dots, i-1 \\
 T'_i &= T_i - \{j\} & B'_i &= B_i - \{k\} \\
 T'_{i+1} &= \{j\} & B'_{i+1} &= \{k\} \\
 T'_l &= T_{l-1} & B'_l &= B_{l-1} & l &= i+2, \dots, m+1
 \end{aligned}$$

To illustrate, consider the search tree in Fig. 13.12(a) which enumerates all

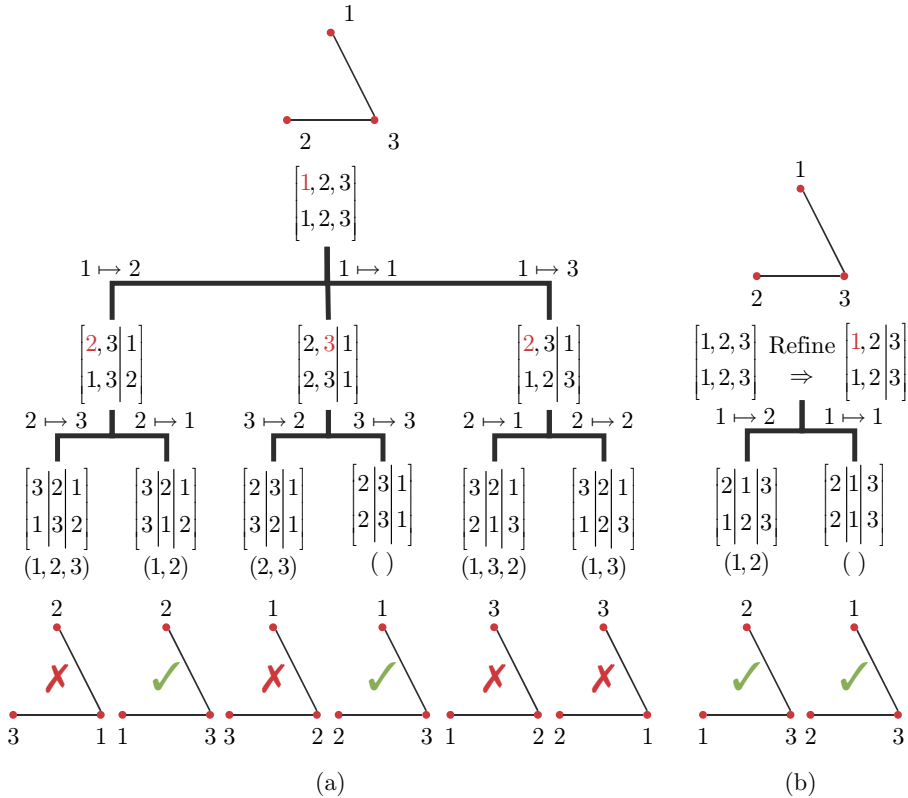


Figure 13.12. Search trees for the automorphisms of a 3-vertex “line” graph. The target vertex (“decision variable”) at each tree node is highlighted in red. (a) without partition refinement. (b) with partition refinement.

permutations of $V = \{1, 2, 3\}$ and checks which are valid symmetries of the indicated 3-vertex 2-edge graph. Each node of the search tree corresponds to an OPP which is the root of a subtree obtained by mapping a target vertex in all possible ways. For example, the unit OPP at the root of the search tree is extended into a 3-way branch by mapping target vertex 1 to 2, 1, and 3. It is important to point out that the choice of target vertex at each tree node and the order in which each of its possible mappings are processed does not affect the final set of permutations produced at the leaves of the search tree. It does, however, alter the order in which these permutations are produced. Note that valid permutations, i.e. permutations that satisfy the graph’s edge relation, can be viewed as satisfying assignments whereas invalid ones are analogous to SAT conflicts.

The permutation search tree can be pruned significantly by performing *partition refinement* [AHU74, DLMS04, McK81] before selecting and branching on a target vertex. This is analogous to Boolean constraint propagation in the SAT space and is standard in all algorithms for graph automorphism and canonical labeling. In the present context, partition refinement is applied *simultaneously* to the top and bottom partitions of the current OPP. This is illustrated in Fig. 13.12(b) where vertex 3 is split from vertices 1 and 2 because it has a

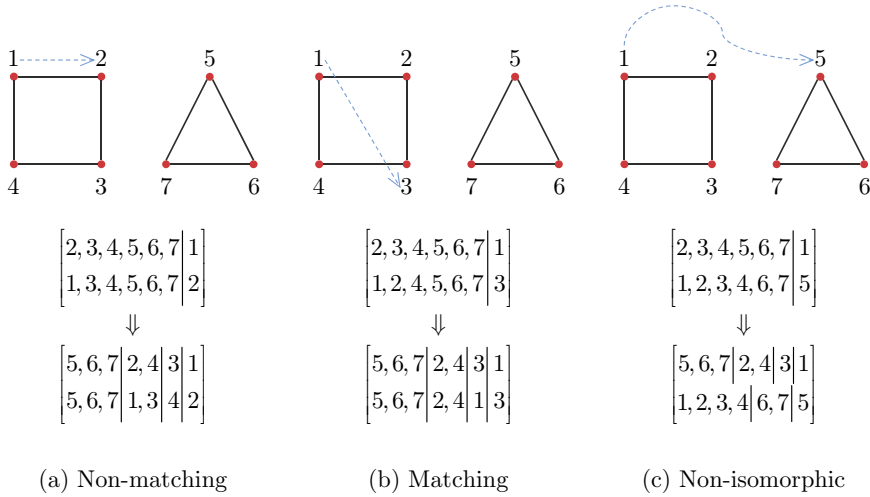


Figure 13.13. Possible outcomes of three different mappings and subsequent refinement of the unit OPP $\left[\begin{array}{c} 1,2,3,4,5,6,7 \\ 1,2,3,4,5,6,7 \end{array} \right]$. In (a) mapping vertex 1 to vertex 2 leads to a non-matching OPP that requires further branching before discovering a potential symmetry. In (b) mapping vertex 1 to vertex 3 leads to a matching OPP that corresponds to the 12 valid permutations $(1\ 3) \circ S_2(\{2,4\}) \circ S_3(\{5,6,7\})$ all of which require the swap of vertices 1 and 3. In this case, we can prune further branching since the matching non-singleton cells can be viewed as “don’t care’s.” In (c), mapping vertex 1 to vertex 5 causes the top and bottom partitions to split non-isomorphically into 4 and 3 cells, respectively, allowing us to immediately conclude that there are no valid permutations in this branch.

different degree.

As in SAT search, partition refinement is invoked after each mapping decision to determine the consequences of that decision. In the SAT case, the consequences are implied assignments to other variables that move the search from its current *decision* state to one of the following three states:

- A new decision state if there are still unassigned variables after propagation.
- A *solution* state if all variables have been assigned; this would be a satisfying assignment.
- A *conflict* state in which a variable is assigned opposing values.

The consequences of a decision in the permutation search tree is a refinement of an isomorphic OPP that moves the search from its current decision state to one of the following three states:

- A new decision state if the resulting OPP is non-matching.
- A solution state if the resulting OPP is matching. Unlike the SAT case, this state corresponds to a partial assignment that represents many solutions¹⁰.

¹⁰This is similar to how the original Grasp SAT solver [MSS96, MSS99] terminated the search: it stopped when it detected that all clauses are satisfied by the current partial assignment. Modern CDCL SAT solvers that employ Chaff’s two-watched literal scheme [MMZ⁺01] terminate

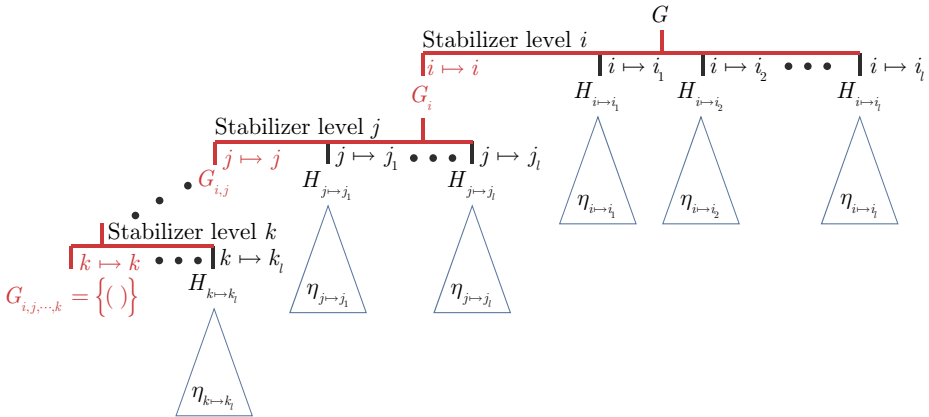


Figure 13.14. Structure of the **saucy** permutation search tree. The highlighted branching levels correspond to subgroup stabilizers. The triangles correspond to permutation sets that are either empty or that represent cosets of the stabilizer subgroup in the left-most branch.

- A conflict state if the corresponding OPP is non-isomorphic.

These three cases are illustrated in Fig. 13.13 for an example 7-vertex graph. The occurrence of matching and non-isomorphic OPPs during the search for automorphisms can be used to define two OPP-based pruning rules that significantly reduce unnecessary exploration of the permutation search tree:

- **Matching OPP pruning** which avoids further branching by mapping matching cells identically and returning an automorphism based on the singleton cells.
- **Non-isomorphic OPP pruning** which indicates that all permutations in the subtree rooted at that node violate the graph’s edge relation.

13.8.3. Group-Theoretic Pruning: Finding a Group’s Generators

Enumerating all automorphisms of a given graph is, in general, intractable but also unnecessary. Instead, a graph automorphism algorithm simply needs to find a set of irredundant generators for the graph’s automorphism group. This is achieved by applying two rules for pruning the permutation search tree that are anchored in group theory: *coset* pruning and *orbit* pruning. These two rules are orthogonal to, and work in concert with, the above-mentioned OPP-based pruning rules.

To enable coset and orbit pruning, branching decisions along the left-most path in the permutation tree are restricted to be identity mappings. This results in a decomposition of the graph’s automorphism group G into a sequence of stabilizer subgroups ending in the identity permutation. As shown in Fig. 13.14, this restriction structures the tree into a sequence of stabilizer levels corresponding to the sequence of stabilized vertices along the left-most path. Each level is

when all variables are assigned.

a multi-way branch whose left-most node is a stabilizer subgroup and whose remaining nodes are possible cosets of that subgroup.

For example, stabilizer level i in Fig. 13.14 corresponds to the multi-way branch that maps vertex i to vertices i, i_1, i_2, \dots, i_l in that order. The left-most node at this level represents the permutation subset corresponding to the mapping $i \mapsto i$, namely the subgroup G_i that stabilizes i . The other nodes at this level correspond to the mappings $i \mapsto i_x$ for $x = 1, \dots, l$ and represent permutation subsets that will be denoted as $H_{i \mapsto i_x}$.

Now assume that Z is a set of irredundant generators for G_i . A set of generators for the parent group G can be obtained by augmenting Z with a *single* representative from each coset of G_i . This set may, however, contain redundant generators that must be eliminated with the aid of the orbit partition.

To find coset representatives, we must solve up to l independent problems where problem $i \mapsto i_x$ seeks to determine whether the set of permutations $H_{i \mapsto i_x}$ is a coset of G_i . This is accomplished by searching $H_{i \mapsto i_x}$ for a *single permutation* that satisfies the graph edge relation, i.e., a permutation that is an automorphism of the graph. If no such permutation exists, then $H_{i \mapsto i_x}$ is “unsatisfiable”, i.e., it is not a coset of G_i . This problem is remarkably similar, structurally, to the problem of finding a satisfying assignment to a CNF formula or proving that no such assignment exists.

Let permutation $\eta_{i \mapsto i_x}$ denote a solution to problem $i \mapsto i_x$. Clearly, $\eta_{i \mapsto i_x}$ serves as a coset representative for $H_{i \mapsto i_x}$ and is added to the set of generators for G . Additionally, the orbit partition is updated by merging the orbits of i and i_x . Thus, if the orbit of i_x contains vertex i_y with $y > x$, then problem $i \mapsto i_y$ can be skipped since its corresponding coset must necessarily contain redundant generators.

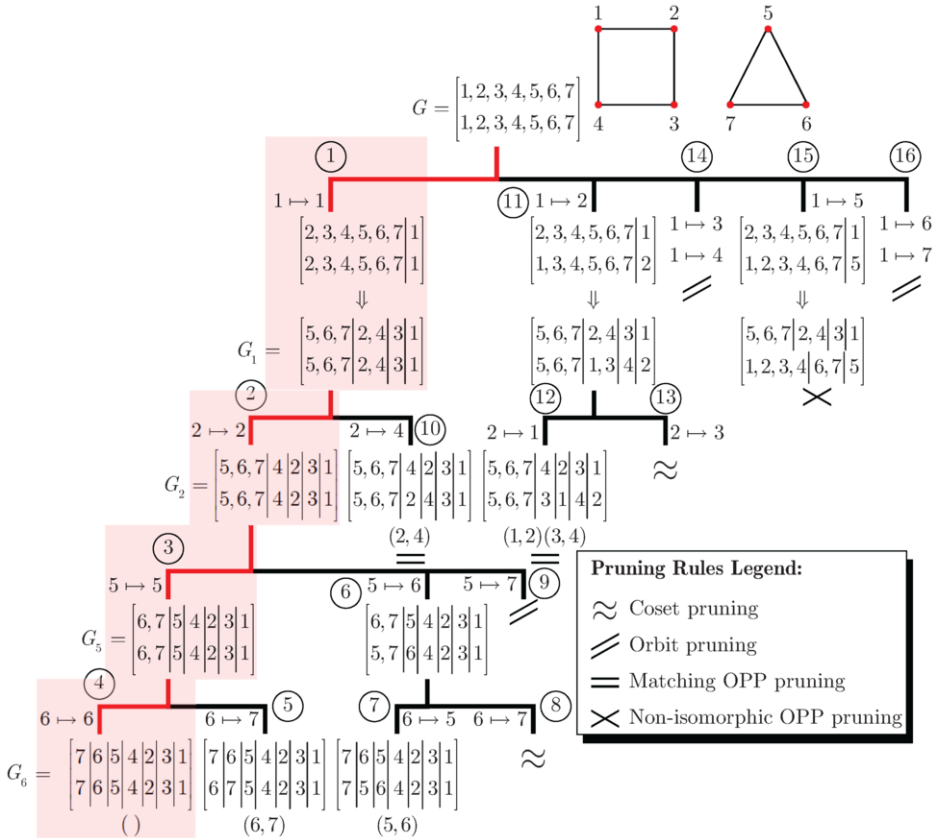
To summarize, the two group-theoretic pruning rules that guarantee the production of an irredundant set of group generators are:

- **Coset pruning** which terminates the search in a coset subtree as soon as a coset representative is found.
- **Orbit pruning** which avoids searching the subtree of $H_{i \mapsto i_x}$ if i_x is already in i 's orbit or in the orbit of a vertex i_y that was previously shown to lead to a violation of the graph's edge relation under the mapping $i \mapsto i_y$.

Finding a coset representative for problem $H_{i \mapsto i_x}$, or proving that the problem is not a coset, is greatly facilitated by the two OPP-based pruning rules. Specifically, a coset representative is found as soon as a node in the search tree becomes matching. When this happens at a shallow level in the tree, a potentially exponential amount of work is avoided. Similarly, encountering a non-isomorphic node eliminates the need for further branching. When this happens at the level of stabilizer subgroup G_i , i.e., when attempting to solve problem $H_{i \mapsto i_x}$, it immediately indicates that this problem is not a coset of G_i .

13.8.4. Putting it All Together

We now have all the ingredients needed to sketch the **saucy** algorithm. Given a colored graph, the algorithm produces a) a set Z of (no more than $n - 1$)



Initialization: $\pi = \{1 | 2 | 3 | 4 | 5 | 6 | 7\}$, $Z = \emptyset$

1. Fix vertex 1 and refine
2. Fix vertex 2
3. Fix vertex 5
4. Fix vertex 6; $G_6 = \{()\}$
5. Search for representative of coset $H_{6 \rightarrow 7}$;
 $Z = \{(6, 7)\}$; $\pi = \{1 | 2 | 3 | 4 | 5 | 6, 7\}$; $|G_5| = |G_6| \cdot |G_6| = 1 \cdot 2 = 2$
6. Search for representative of coset $H_{5 \rightarrow 6}$
7. Found representative of coset $H_{5 \rightarrow 6}$;
 $Z = \{(6, 7), (5, 6)\}$; $\pi = \{1 | 2 | 3 | 4 | 5, 6, 7\}$
8. **Coset pruning**: no need to explore since we have already found a coset representative for $H_{5 \rightarrow 6}$
9. **Orbit pruning**: no need to explore since 7 is already in the orbit of 5
 $|G_2| = |G_5| \cdot |G_5| = 2 \cdot 3 = 6$
10. Search for representative of coset $H_{2 \rightarrow 4}$;
Matching OPP pruning: found representative of coset $H_{2 \rightarrow 4}$
 $Z = \{(6, 7), (5, 6), (2, 4)\}$; $\pi = \{1 | 3 | 2, 4 | 5, 6, 7\}$; $|G_1| = |G_2| \cdot |G_2| = 6 \cdot 2 = 12$
11. Search for representative of coset $H_{1 \rightarrow 2}$
12. **Matching OPP pruning**: found representative of coset $H_{1 \rightarrow 2}$.
 $Z = \{(6, 7), (5, 6), (2, 4), (1, 2)(3, 4)\}$; $\pi = \{1, 2, 3, 4 | 5, 6, 7\}$
13. **Coset pruning**: no need to explore since we have already found a coset representative for $H_{1 \rightarrow 2}$
14. **Orbit pruning**: no need to explore since 3 and 4 are already in the orbit of 1
15. **Non-isomorphic OPP pruning**: 1 cannot map to 5
16. **Orbit pruning**: no need to explore since 6 and 7 are already in the orbit of 5
 $|G| = |G_1| \cdot |G_1| = 12 \cdot 4 = 48$

Figure 13.15. Search tree for graph automorphisms of the “square and triangle” graph and relevant computations at each node. The shaded region corresponds to subgroup decomposition.

irredundant generators for the graph's automorphism group G , b) the group's orbit partition π , and c) the order $|G|$ of the group .

The algorithm starts out with an empty set of generators $Z = \emptyset$ and an orbit partition where each graph vertex i is in its own cell, $G_i = \{i\}$. The algorithm then proceeds with a depth-first traversal of the tree to obtain a stabilizer subgroup decomposition of G . This is followed by repeatedly processing each stabilizer level, in reverse order, to compute Z , π , and $|G|$.

This computation is best understood by considering two adjacent stabilizer levels i and j in the permutation search tree (see Fig. 13.14). Let Z_i (resp. π_i) and Z_j (resp. π_j) denote the set of generators (resp. orbit partitions) at levels i and j . In addition, let B_j denote the set of vertices that result in non-isomorphic OPPs when j is mapped to any one of them; these are “bad” vertices. The computation at stabilizer level j can now be summarized as follows:

1. $Z_i = Z_j$; $\pi_i = \pi_j$; $B_j = \emptyset$; // Initialize
2. for $x = 1$ to l // Process mappings of j
3. if $\neg[(j_x \in Gj) \vee (\exists j_y \in B_j : j_x \in Gj_y)]$ // Orbit pruning
4. if SAT($H_{j \rightarrow j_x}$) { // Found symmetry $\eta_{j \rightarrow j_x}$
5. $Z_i = Z_i \cup \{\eta_{j \rightarrow j_x}\}$ // Update generators
6. Update π_i by merging Gj and Gj_x } // Update orbit partition
7. else $B_j = B_j \cup \{j_x\}$ // Record bad mapping
8. $|G_i| = |G_j| \cdot |G_j|$ // Compute order of G_i

The check on line 3 implements orbit pruning by ensuring that the mapping $j \mapsto j_x$ can be skipped if either j_x is already in j 's orbit or it is in the orbit of a vertex in j 's bad set. The SAT check on line 4 utilizes the two OPP pruning rules to quickly find an automorphism or to prove that none exist for the given mapping.

A trace of this (informally presented) algorithm illustrating all four pruning rules is shown in Fig. 13.15. Fig. 13.16 shows its run time performance as implemented in the **saucy** symmetry detection tool on the complete set of 1183 SAT 2009 competition benchmarks. As the figure shows, there is a weak trend towards larger run times for larger graphs. However, run time seems to also depend on other attributes of a graph besides its absolute size (number of vertices.) Overall, the algorithm seems to be quite scalable: it solved the largest benchmark **post-cbmc-zfcp-2.8-u2-noholes**, an application instance with about 11 million variables and 33 million clauses, whose graph encoding has over 32 million vertices, in about 231 seconds. It is also extremely fast, finishing in less than one second on 93% (1101) of all benchmarks.

13.9. Summary and a Look Forward

This chapter was concerned with recognizing symmetry in CNF formulas and exploiting it in the context of satisfiability checking. The particular approach we described for utilizing symmetry in SAT solving is sometimes referred as *static symmetry breaking* because it modifies the input to the SAT solver rather than modifying the SAT solver itself. Given a CNF formula $\varphi(X)$, static symmetry breaking consists of the following steps:

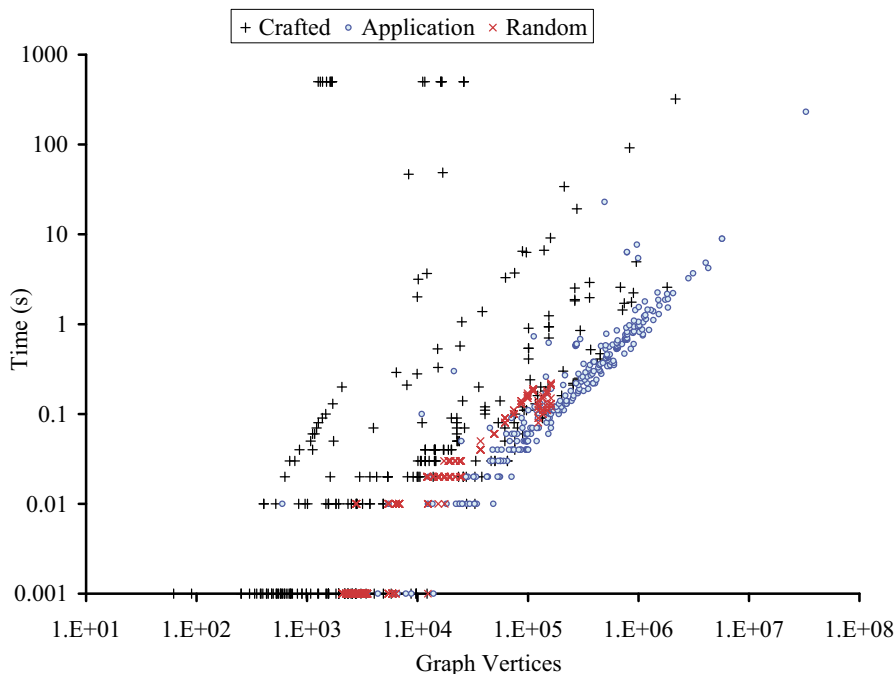


Figure 13.16. saucy 2.1 run time, in seconds, as a function of graph size for the SAT 2009 competition benchmarks. A time-out of 500 seconds was applied.

1. Converting the formula to a colored graph whose symmetries are isomorphic to the symmetries of the formula.
2. Computing the automorphism group G_φ of the graph and returning a set of generators $\hat{G}_\varphi \subset G_\varphi$ for it using a suitable graph automorphism program.
3. Mapping the returned generators back to symmetries of the formula.
4. Constructing an appropriate symmetry-breaking predicate $\rho(\hat{G}_\varphi; X)$ in CNF and conjoining it to the formula.
5. Checking $\varphi(X) \wedge \rho(\hat{G}_\varphi; X)$ using a suitable SAT solver.

The attractiveness of this approach stems from the fact that it can be used with *any* SAT solver and can, thus, automatically benefit from improvements in SAT solving technology. On the other hand, the method has a number of drawbacks that can limit its effectiveness in some cases. The method's inherent disadvantage is that the SBP created in step 4 relates to the *global symmetries* of the original formula. During SAT search, particular partial assignments to the variables may give rise to additional *local symmetries*, also known as *conditional symmetries* [BS07], that offer the possibility of further pruning of the search space. A pre-processing approach cannot utilize such symmetries. Computationally, even after applying the optimizations described in Sec. 13.7.1, the size of the SBP (expressed as a CNF formula) might still be too large to be effectively handled by a SAT solver. Furthermore, the pruning ability of the SBP depends on the generator set used to derive it and, as we have seen experimentally, can vary widely. Finally, the assumed, and essentially arbitrary, ordering of the variables to create the SBP

may be at odds with the SAT solver's branching strategy; this limitation, however, is not as severe as the other two since modern SAT solvers employ adaptive decision heuristics that are insensitive to the initial static ordering of the variables. It may be useful, however, to explore orderings on the truth assignments that are not lexicographic. For example, orderings in which successive assignments have a Hamming distance of 1 (i.e., differ in only one bit position) might yield better SBPs because of better clustering of "least" assignments. Despite these shortcomings, static symmetry breaking is currently the most effective approach for exploiting symmetry in SAT search.

An alternative approach is to break symmetries *dynamically* during the search. In fact, one of the earliest references to utilizing symmetry in SAT advocated such an approach [BFP88]. However, except for a brief follow-up to this work in [LJPJ02], the early contributions of Benhamou et al in [BS92, BS94], and the more recent contribution of Sabharwal in [Sab05], there is practically little mention in the open literature of dynamic symmetry breaking in SAT. The situation is completely different in the domain of constraint satisfaction problems (CSP) where breaking symmetries during search has been a lively research topic for the past few years. One conjecture that might explain this anomaly is that modern implementations of conflict-driven backtrack SAT solvers are based on highly-optimized and finely-tuned search engines whose performance is adversely affected by the introduction of expensive symmetry-related computations in their innermost kernel.

The potential benefits of dynamic symmetry breaking are too compelling, however, to completely give up on such an approach. These benefits include the ability to identify local symmetries, and the possibility of boosting the performance of conflict-driven learning by recording the symmetric equivalents of conflict-induced clauses. Dynamic symmetry breaking is also appealing because it avoids the creation, as in static symmetry breaking, of potentially wasteful symmetry breaking clauses that drastically increase the size of the CNF formula while only helping marginally (or not at all) in pruning the search space. To realize these benefits, however, the integration of symmetry detection and symmetry breaking must be considered carefully. An intriguing possibility is to *merge* the graph automorphism and SAT search engines so that they can work simultaneously on the CNF formula as the search progresses. The recent improvements in graph automorphism algorithms, tailored specifically to the sparse graphs that characterize CNF formulas, suggest that this may now be a practical possibility. Further advances, including incremental updating of the symmetry group (through its generators) in response to variable assignments, are necessary, however, to make such a scheme effective on large CNF instances with many local symmetries. In short, this is an area that is ripe for original research both in the graph automorphism space as well as in the SAT search space.

13.10. Bibliographic Notes

13.10.1. Theoretical Considerations

The study of symmetry in Boolean functions has a long history. Early work was primarily concerned with counting the number of symmetry types of Boolean functions of n variables. Assuming invariance under variable permutation and/or complementation, the 2^{2^n} possible functions of n variables can be partitioned into distinct equivalence classes. Slepian [Sle54] gave a formula for the number of such classes as a function of n . Slepian also related this to earlier work by Pólya [Pól40] and Young [You30], and noted that the group of permutations and/or complementations is isomorphic to the hyperoctahedral group (the automorphism group of the hyperoctahedron in n -dimensional Euclidean space) as well as to the group of symmetries of the n -dimensional hypercube. These investigations were further elaborated by Harrison [Har63] who gave formulas for separately counting the number of equivalence classes of Boolean functions under three groups: complementation, permutation, and combined complementation and permutation. He also pointed out that the combined group is the semi-direct, rather than the direct, product¹¹ of the other two. More recently, Chen [Che93] studied the *cycle structure* of the hyperoctahedral group by representing it using *signed permutations*, i.e., permutations on $\{1, 2, \dots, n\}$ “with a + or – sign attached to each element $1, 2, \dots, n$.”

13.10.2. Symmetry in Logic Design

Symmetry of Boolean functions was also studied in the context of logic synthesis. Motivated by the desire to create efficient realizations of relay circuits for telephone switching exchanges, Shannon [Sha38, Sha49] developed a notation to describe and logically manipulate symmetric functions based on their so-called “a-numbers.” Specifically, Shannon showed that a function which is symmetric in all of its variables (i.e., a function which remains invariant under all permutations of its variables) can be specified by a subset of integers from the set $\{0, 1, \dots, n\}$. Such functions were termed *totally symmetric* in contrast with *partially symmetric* functions that remain invariant under permutation of only a *subset* of their variables. For example, $f(a, b, c) = ab + ac + bc$ is a totally symmetric function specified as $S_{\{2,3\}}(a, b, c)$ and interpreted to mean that f is equal to 1 when exactly 2 or exactly 3 of its variables are 1. The notation extends naturally when some variables need to be complemented before being permuted. Thus $f(a, b, c) = ab'c + a'bc + a'b'c'$ is specified as $S_{\{0,1\}}(a, b, c')$ meaning that f is equal to 1 when exactly 0 or exactly 1 of the literals from the set $\{a, b, c'\}$ is equal to 1. The specification of partially symmetric functions is more involved. Let $f(X, Y)$ be such a function where $\{X, Y\}$ is a partition of its variables such that the function remains invariant under all permutations of the X variables.

¹¹A discussion of direct and semi-direct group products will take us too far afield. For current purposes, it is sufficient to note that a direct product is a special type of a semi-direct product. Specifically, the direct product of two groups G and H is the group whose underlying set is $G \times H$ and whose group operation is defined by $(g_1, h_1)(g_2, h_2) = (g_1g_2, h_1h_2)$ where $g_1, g_2 \in G$ and $h_1, h_2 \in H$.

Using the elementary symmetric functions¹² $S_{\{i\}}(X)$ for $i = 0, \dots, |X|$ as an orthonormal basis, f can be specified as:

$$f(X, Y) = \sum_{0 \leq i \leq |X|} S_{\{i\}}(X) R_i(Y)$$

where $R_i(Y)$ is the *residual* function corresponding to $S_{\{i\}}(X)$.

Shannon's definition of symmetry yields a partition of a function's literals such that literals in a given cell of the partition can be permuted arbitrarily without causing the function to change. A variety of schemes were subsequently developed to derive this partition. All of these schemes are based on finding pairs of equivalent literals using the following "Boole/Shannon" expansion:

$$f(\dots, x, \dots, y, \dots) = x'y'f_{x'y'} + x'yf_{x'y} + xy'f_{xy'} + xyf_{xy}$$

where $f_{x'y'}, \dots, f_{xy}$ are the *cofactors* of f with respect to the indicated literals [BCH⁺82]. For example, $f_{x'y} = f(\dots, 0, \dots, 1, \dots)$, i.e., f is restricted by the assignment of 0 to x and 1 to y . It is now straightforward to see that f remains invariant under a swap of x and y if and only if $f_{x'y} = f_{xy'}$. Similarly, f will be invariant under a swap of x and y' (resp. x' and y) if and only if $f_{x'y'} = f_{xy}$. In cycle notation,

$$\begin{aligned} (f_{x'y} = f_{xy'}) &\Leftrightarrow (x, y) \\ (f_{x'y'} = f_{xy}) &\Leftrightarrow (x, y') \end{aligned}$$

Larger sets of symmetric literals can now be built from symmetric pairs using transitivity. The computational core in all of these schemes is the check for equivalence between the cofactor functions. Early work [Muk63] employed *decomposition charts* which, essentially, are truth tables in which the four combinations of the variable pair being checked for equivalence are listed row-wise whereas the combinations of the remaining variables are listed column-wise. Equality of cofactors is now detected as identical rows in the charts. Such a scheme, obviously, is not scalable and subsequent symmetry detection algorithms mirrored the evolution, over time, of more efficient representations for Boolean functions as cube lists (that encode functions in SOP form) [DS67], in terms of various spectral transforms [EH78, RF98], using Reed-Muller forms [TMS94], and finally as binary decision diagrams (BDDs) [MMW93, PSP94]. Most recently, the problem of detecting the symmetries of large Boolean functions was addressed by employing a portfolio of representations and algorithms including circuit data structures, BDDs, simulation, and (interestingly) satisfiability checking [ZMBCJ06].

It is important to note that the symmetries we have been discussing in this brief historical review are *semantic* rather than syntactic, i.e., symmetries that are independent of a function's particular representation. Detection of these symmetries, thus, requires some sort of *functional* analysis that checks various cofactors for logical equivalence¹³. Furthermore, we may view the resulting partition of a

¹² $S_A(X)$ is an elementary symmetric function if $|A| = 1$, i.e., the set of a-numbers A is a singleton.

¹³As mentioned in Sec. 13.4, semantic symmetries can also be found by structural analysis of certain CNF representations of a function such as those that include a function's maxterms or its prime implicates. However, since these representations tend to be exponential in the number of variables, they are rarely used to find a function's semantic symmetries.

function's literals into subsets of symmetric literals as an implicit compact representation of the function's symmetries. This is in contrast with the use of a set of irredundant generators to represent a group of symmetries. We should note, however, that a partition of the function's literals may not capture all possible symmetries of the function. This fact was "discovered" in the context of checking the logical equivalence of two functions with unknown correspondence between their inputs [MMM95], and led to augmenting Shannon's *classical symmetries*, which are based on a flat partitioning of a function's literals, with additional *hierarchical symmetries* that involve simultaneous swaps of sets of literals. These symmetries were further generalized in [KS00] where classical *first-order* symmetries (based on variable swaps) formed the foundation for *higher-order* symmetries that involve the swaps of both ordered and un-ordered sets of literals. These two extensions to Shannon's original definition of functional symmetry allowed the detection of many more function-preserving literal permutations in benchmark circuits. Still, such swap-based representations are inherently incapable of capturing arbitrary permutations such as $(a, b, c)(d, f, e)(g, h, m)(i, j, k)$ which is a symmetry of the function $ab + ac + af + aj + bc + be + bk + cd + ci + dk + ej + fi + gi + hj + km$ [Kra01].

13.10.3. Symmetry in Boolean Satisfiability Problems

Whereas interest in semantic symmetries, as noted above, was primarily motivated by the desire to optimize the design of logic circuits or to speed up their verification, interest in syntactic symmetries, i.e., symmetries of CNF formulas, arose in the context of constraint solving, specifically Boolean satisfiability checking. The earliest reference to the potential benefits of invoking symmetry arguments in logical reasoning is traditionally attributed to Krishnamurthy [Kri85] who introduced the notions of global and local "symmetry rules" that can be used to significantly shorten (from exponential to polynomial) the proofs of certain propositions such as the pigeon-hole principle. Another early reference to the use of symmetry in backtrack search is [BFP88] where symmetries are used *dynamically* by the search algorithm to restrict its search to the equivalence classes induced by the formula's symmetries (which are assumed to be given). In the same vein, Benhamou and Sais [BS92, BS94] described a modification to backtrack search that prunes away symmetric branches in the decision tree. Their method for detecting symmetries worked directly on the CNF formula and derived formula-preserving variable permutations incrementally through pair-wise substitutions. In [Sab05], Sabharwal described an interesting modification to the zChaff [MMZ⁺01] SAT solver that allows it to branch on a set of symmetric variables rather than on single variables. For instance, given a set $\{x_1, x_2, \dots, x_k\}$ of k symmetric variables, a $k + 1$ -way branch sets x_1, \dots, x_i to 0 and x_{i+1}, \dots, x_k to 1 for each $i \in [0, k]$. This reduces the number of partial assignments that must potentially be explored from 2^k to $k + 1$. The identification of symmetric variables is assumed to be available from high-level descriptions of a problem (e.g., variables denoting pigeons in a pigeon-hole are all symmetric) and provided to the solver as an additional input.

The first complete exposition of the use of symmetries in a pre-processing step to prune the search space of SAT solvers must, however, be credited to

Crawford et al. [Cra92, CGLR96]. In [Cra92], Crawford laid the theoretical foundation for reasoning by symmetry by showing that symmetry detection can be polynomially reduced to the problem of colored graph automorphism. The subsequent paper [CGLR96] detailed the basic flow of symmetry detection and breaking and introduced the following computational steps which, with some modification, still constitute the elements of the most successful approach, at least empirically, for the application of symmetry reasoning in SAT. Given a CNF formula:

1. Convert the formula to a colored graph whose automorphisms correspond to the symmetries of the formula.
2. Identify a set generators of the graph automorphism group using the **nauty** graph automorphism package [McK, McK81].
3. Using the generators, create a *lex-leader*¹⁴ symmetry-breaking predicate from one of the following alternative sets of symmetries:
 - just the generators.
 - a subset of symmetries obtained from a truncated *symmetry tree* constructed from the generators.
 - a set of random group elements obtained from the generators.

The symmetry tree data structure in this flow was proposed as a mechanism to reduce the observed duplication in the SBP for problems with a large number of symmetries. However, except for some special cases (e.g., the full symmetry group of order $n!$ has a pruned symmetry tree with n^2 nodes), the size of the symmetry tree remains exponential in the number of variables.

The symmetry detection and breaking approach we described in Sec. 13.6 and Sec. 13.7 is based on [ARMS02, AMS03, ASM03] which extended Crawford's methodology in the following important ways:

- Crawford's graph construction in [CGLR96] used three vertex colors (for clauses, positive literals, and negative literals, respectively) and was, thus, limited to discovering variable symmetries. By using one color for both positive and negative literals, it was shown in [ARMS02] that additional symmetries can be found, namely value symmetries and their composition with variable symmetries. The restriction to just value symmetries by the use of a distinct color for each literal pair, as described in Sec. 13.6, is a further extension of these ideas that shows the versatility of the colored graph construction as a mechanism for studying the symmetries in a CNF formula.
- The size of Crawford's lex-leader SBP for breaking a single symmetry is quadratic in the number of variables. The optimizations introduced in [ARMS02, AMS03], and particularly in [ASM03], reduced this dependence to linear in a much smaller number that reflects the cycle structure of a symmetry as well as the presence of phase shifts in it. The relaxed permutation predicate in (13.32) represents a further optimization that

¹⁴The term lex-leader indicates a lexicographic ordering of the truth assignments that is induced by an assumed ordering of the variables; it is the same as the numeric ordering in (13.5) corresponding to the interpretation of the truth assignments as unsigned integers.

shrinks the size of the SBP from $14|I|$ to $9|I|$ where I is the index set of irredundant bits defined by (13.26).

- The use of edges to represent binary clauses was suggested by Crawford to improve the efficiency of graph automorphism. The possibility that this optimization to the colored graph construction may produce spurious symmetries was noted in [ARMS03] and attributed to the presence of circular chains of implications in the CNF formula. Alternative constructions were also offered in [ARMS03] to deal with this problem, as well as suggestions for how to filter out the spurious symmetries when they arise.
- Finally, the work in [ARMS02, AMS03, ASM03] provided the first empirical large-scale validation of the feasibility of static symmetry breaking. This approach was validated empirically on a large number of different benchmark families.

Recent applications of this general symmetry breaking approach included its use in decision and optimization problems that involve pseudo-Boolean constraints (linear inequalities on Boolean variables) [ARSM07], and in deciding the satisfiability of quantified Boolean formulas [AJS07].

13.10.4. Symmetry in Constraint Satisfaction Problems

Research on the use of symmetry in constraint programming is too voluminous to adequately cover in a few paragraphs. Instead, we refer the reader to the excellent recent (and largely current) survey in [GPP06]¹⁵ and briefly highlight a few connections to similar issues in the context of Boolean satisfiability.

One of the questions that has preoccupied the constraint programming community over the past several years has been definitional: what exactly are the symmetries of a constraint programming problem? This question was largely settled in [CJJ⁺05] which provided a comprehensive taxonomy that encompassed most of the previous notions of CSP symmetry. In a nutshell, the type of symmetry in a CSP is classified based on:

- What is being permuted:
 - variables (yielding *variable* symmetries)
 - values (yielding *value* symmetries)
 - variable/value pairs
- What is invariant:
 - set of solutions (yielding *solution* symmetries)
 - set of constraints (yielding *constraint* symmetries)

This classification is essentially the same as the one we used to categorize the symmetries of CNF formulas. Specifically, a CNF's semantic and syntactic symmetries correspond, respectively, to a CSP's solution and constraint symmetries. Furthermore, viewing a CSP's variable/value pair as a *literal*, we note that a CNF's variable, value, and mixed symmetries have their corresponding CSP counterparts. The similarity extends to the graph constructions used to extract the symmetries of a problem instance. For example, the constraint symmetries of a

¹⁵This survey also has pointers to key references on the use of symmetry in integer programming, planning, theorem proving, and model checking.

CSP instance correspond to automorphisms of the associated *microstructure* [J93] which is the analog of the CNF formula graph we used in Sec. 13.6 to find the formula's syntactic symmetries.

There are some important differences, however. Symmetries tend to be easier to spot and describe in the CSP specifications of many problems than in the corresponding CNF encodings of these problems (the pigeon-hole problem being an obvious example.) There is room, thus, for hybrid approaches, along the lines of [Sab05], that combine high-level specifications of a problem's symmetries with a modern SAT solver's powerful search algorithms. Furthermore, many of the techniques for dynamic symmetry breaking that have been studied extensively in the CSP domain have yet to be fully explored in the CNF SAT context.

13.10.5. Graph Automorphism

Crawford's proposal in [CGLR96] for detecting and breaking the symmetries of CNF formulas included the use of McKay's graph isomorphism package, **nauty** [McK, McK81]. **nauty** had, by then, been the dominant publicly-available tool for computing automorphism groups, as well as for canonical labeling, of graphs. It was, thus, the natural choice for symmetry detection in subsequent extensions to Crawford's work in both the SAT and CSP domains. However, it quickly became clear that **nauty** was not optimized for the large sparse graphs constructed from CNF formulas that model design or verification problems. Specifically, the significant reductions in SAT search time that became possible because of SBP pruning were more than offset by **nauty**'s runtime for the detection of symmetries. This prompted the development of **saucy** [DLSM04] whose data structures were designed to take advantage of the sparsity in the graphs constructed from CNF formulas. **saucy** achieved significant speed-ups over **nauty** on such graphs and established the viability, at least for certain classes of problems, of the static symmetry detection and breaking approach. More enhancements along these lines were recently proposed in [JK07] which introduced the **bliss** tool for canonical labeling of large and sparse graphs. The latest offering in this space is a major enhancement to **saucy** that not only takes advantage of the sparsity of the input graph but also of the symmetry generators themselves [DSM08].

13.10.6. Canonical Labeling

While the main concern in this chapter is symmetry detection to speed up SAT solving, the accidental discovery of fast graph automorphism algorithms that de-link symmetry detection from canonical labeling opened the possibility of also re-thinking the architecture of canonical labeling tools. Katebi et al. [KSM12] showed that by carefully integrating **saucy** with **bliss**, canonical labeling can be done much faster. Specifically, by synchronizing the stabilizer subgroup decomposition between **saucy** and **bliss**, symmetry detection is quickly carried out by **saucy** which it forwards to **bliss** as orbit partitions at each stabilizer level. This eliminates the need for **bliss** to perform any symmetry processing since it now has all the information necessary to quickly find a canonical labeling.

Acknowledgments

Many people helped in shaping the content of this chapter. I want to thank Igor Markov for nudging me to explore group theory as a way to describe and analyze symmetries of Boolean functions. Fadi Aloul was tremendously helpful in providing the experimental data for the mixed, variable, and value symmetries of CNF formulas. Paul Darga helped improve the description of the basic graph automorphism algorithm. Mark Liffiton provided the examples used to illustrate the various concepts. Zaher Andraus provided detailed comments from a careful reading of the manuscript. Hadi Katebi and Paolo Codenotti contributed to the development of the latest version of **saucy** and its integration with **bliss** to enable fast canonical labeling. I would also like to thank my editor Toby Walsh for his insightful feedback. This work was completed during the author's sabbatical leave at Carnegie Mellon in Qatar and was supported, in part, by the National Science Foundation under ITR grant No. 0205288.

References

- [AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- [AJS07] G. Audemard, S. Jabbour, and L. Sais. Symmetry breaking in quantified boolean formulae. In *Proc. International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 2262–2267, 2007.
- [AMS03] F. A. Aloul, I. L. Markov, and K. A. Sakallah. Shatter: Efficient symmetry-breaking for boolean satisfiability. In *Proc. 40th IEEE/ACM Design Automation Conference (DAC)*, pages 836–839, Anaheim, California, 2003.
- [ARMS02] F. A. Aloul, A. Ramani, I. Markov, and K. A. Sakallah. Solving difficult sat instances in the presence of symmetry. In *Proc. 39th IEEE/ACM Design Automation Conference (DAC)*, pages 731–736, New Orleans, Louisiana, 2002.
- [ARMS03] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. Solving difficult instances of boolean satisfiability in the presence of symmetry. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(9):1117–1137, 2003.
- [ARSM07] F. A. Aloul, A. Ramani, K. A. Sakallah, and I. L. Markov. Symmetry breaking for pseudo-boolean constraints. *ACM Journal of Experimental Algorithms*, 12(Article No. 1.3):1–14, 2007.
- [ASM03] F. A. Aloul, K. A. Sakallah, and I. L. Markov. Efficient symmetry breaking for boolean satisfiability. In *Proc. 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 271–282, Acapulco, Mexico, 2003.
- [BCH⁺82] R. K. Brayton, J. D. Cohen, G. D. Hachtel, B. M. Trager, and D. Y. Y. Yun. Fast recursive boolean function manipulation. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 58–62, Rome, Italy, 1982.

- [BFP88] C. A. Brown, L. Finkelstein, and P. W. Purdom. Backtrack searching in the presence of symmetry. In T. Mora, editor, *6th International Conference on Applied Algebra, Algebraic Algorithms and Error Correcting Codes*, pages 99–110, 1988.
- [BS92] B. Benhamou and L. Saïs. Theoretical study of symmetries in propositional calculus and applications. *Lecture Notes In Computer Science*, 607:281–294, 1992. 11th International Conference on Automated Deduction (CADE-11).
- [BS94] B. Benhamou and L. Saïs. Tractability through symmetries in propositional calculus. *Journal of Automated reasoning*, 12:89–102, 1994.
- [BS07] B. Benhamou and M. R. Saidi. Local symmetry breaking during search in cps. *Lecture Notes in Computer Science*, LNCS 4741:195–209, 2007.
- [Can01] E. R. Canfield. Meet and join within the lattice of set partitions. *The Electronic Journal of Combinatorics*, 8:1–8, #R15, 2001.
- [CBB00] B. Cheswick, H. Burch, and S. Branigan. Mapping and visualizing the internet. In *USENIX Annual Technical Conference*, pages 1–13, 2000.
- [CGLR96] J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Principles of Knowledge Representation and Reasoning (KR'96)*, pages 148–159, 1996.
- [Che93] W. Y. C. Chen. Induced cycle structures of the hyperoctahedral group. *SIAM J. Disc. Math.*, 6(3):353–362, 1993.
- [CJJ+05] D. Cohen, P. Jeavons, C. Jefferson, K. E. Petrie, and B. M. Smith. Symmetry definitions for constraint satisfaction problems. *Lecture Notes in Computer Science*, LNCS 3709:17–31, 2005.
- [Cra92] J. Crawford. A theoretical analysis of reasoning by symmetry in first-order logic (extended abstract). In *AAAI-92 Workshop on Tractable Reasoning*, pages 17–22, San Jose, CA, 1992.
- [DIM] DIMACS challenge benchmarks.
<ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/>.
- [DLSM04] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov. Exploiting structure in symmetry detection for cnf. In *Proc. 41st IEEE/ACM Design Automation Conference (DAC)*, pages 530–534, San Diego, California, 2004.
- [DS67] D. L. Dietmeyer and P. R. Schneider. Identification of symmetry, redundancy and equivalence of boolean functions. *IEEE Trans. on Electronic Computers*, EC-16(6):804–817, 1967.
- [DSM08] P. T. Darga, K. A. Sakallah, and I. L. Markov. Faster symmetry discovery using sparsity of symmetries. In *Proc. 45th IEEE/ACM Design Automation Conference (DAC)*, pages 149–154, Anaheim, California, 2008.
- [EH78] C. R. Edwards and S. L. Hurst. A digital synthesis procedure under function symmetries and mapping methods. *IEEE Trans. on Computers*, C-27(11):985–997, 1978.
- [Fra00] J. B. Fraleigh. *A First Course in Abstract Algebra*. Addison Wesley Longman, Reading, Massachusetts, 6th edition, 2000.

- [GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.
- [GPP06] I. P. Gent, K. E. Petrie, and J.-F. Puget. Symmetry in constraint programming. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint programming*. Elsevier, 2006.
- [GT00] R. Govindan and H. Tangmunarunkit. Heuristics for internet map discovery. In *IEEE INFOCOM*, pages 1371–1380, 2000.
- [Har63] M. A. Harrison. The number of transitivity sets of boolean functions. *Journal of the Society for Industrial and Applied mathematics*, 11(3):806–828, 1963.
- [ISP] ISPD 2005 placement competition.
<http://www.sigda.org/ispd2005/contest.htm>.
- [J93] P. Jégou. Decomposition of domains based on the micro-structure of finite constraint-satisfaction problems. In *AAAI'93*, pages 731–736, 1993.
- [JK07] T. Junttila and P. Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *Ninth Workshop on Algorithm Engineering and Experiments (ALENEX 07)*, New Orleans, LA, 2007.
- [Kra01] V. N. Kravets. *Constructive Multi-Level Synthesis by Way of Functional Properties*. PhD thesis, University of Michigan, 2001.
- [Kri85] B. Krishnamurthy. Short proofs for tricky formulas. *Acta Informatica*, 22:253–275, 1985.
- [KS00] V. N. Kravets and K. A. Sakallah. Generalized symmetries in boolean functions. In *Digest of IEEE International Conference on Computer-Aided Design (ICCAD)*, pages 526–532, San Jose, California, 2000.
- [KSM10] H. Katebi, K. A. Sakallah, and I. L. Markov. Symmetry and Satisfiability: An Update. In O. Strichman and S. Szeider, editors, *Thirteenth International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume LNCS 6175, pages 113–127, Edinburgh, July 2010.
- [KSM12] H. Katebi, K. A. Sakallah, and I. L. Markov. Graph Symmetry Detection and Canonical Labeling: Differences and Synergies. In A. Voronkov, editor, *The Alan Turing Centenary Conference (Turing 100)*, volume 10 of *EPiC Series*, pages 181–195, Manchester, UK, June 2012. Best Paper Award.
- [LJPJ02] C. M. Li, B. Jurkowiak, and P. W. Purdom Jr. Integrating symmetry breaking into a dll procedure. In *International Symposium on Boolean Satisfiability (SAT)*, pages 149–155, Cincinnati, OH, 2002.
- [McK] B. D. McKay. nauty user's guide (version 2.2).
<http://cs.anu.edu.au/~bdm/nauty/nug.pdf>.
- [McK81] B. D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.
- [MMM95] J. Mohnke, P. Molitor, and S. Malik. Limits of using signatures for permutation independent boolean comparison. In *Asia South Pacific Design Automation Conference*, pages 459–464, 1995.

- [MMW93] D. Möller, J. Mohnke, and M. Weber. Detection of symmetry of boolean functions represented by robdds. In *International Conference on Computer Aided Design*, pages 680–684, 1993.
- [MMZ⁺01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Design Automation Conference*, pages 530–535, Las Vegas, 2001.
- [MSS96] J. Marques-Silva and K. A. Sakallah. GRASP-A New Search Algorithm for Satisfiability. In *Digest of IEEE International Conference on Computer-Aided Design (ICCAD)*, pages 220–227, San Jose, California, November 1996.
- [MSS99] J. Marques-Silva and K. A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.
- [Muk63] A. Mukhopadhyay. Detection of total or partial symmetry of a switching function with the use of decomposition charts. *IEEE Trans. on Electronic Computers*, EC-12(5):553–557, 1963.
- [NASR01] G.-J. Nam, F. Aloul, K. A. Sakallah, and R. Rutenbar. A comparative study of two boolean formulations of fpga detailed routing constraints. In *International Symposium on Physical Design (ISPD '01)*, pages 222–227, Sonoma, California, 2001.
- [Pó140] G. Pólya. Sur les types des propositions composées. *J. Symbolic Logic*, 5:98–103, 1940.
- [PSP94] S. Panda, F. Somenzi, and B. F. Plessier. Symmetry detection and dynamic variable ordering of decision diagrams. In *International Conference on Computer-Aided Design*, pages 628–631, 1994.
- [Qui52] W. V. Quine. The problem of simplifying truth functions. *Amer. Math. Monthly*, 59(8):521–531, 1952.
- [Qui55] W. V. Quine. A way o simplify truth functions. *Amer. Math. Monthly*, 62(9):627–631, 1955.
- [Qui59] W. V. Quine. On cores and prime implicants of truth functions. *Amer. Math. Monthly*, 66(9):755–760, 1959.
- [RF98] S. Rahardja and B. L. Falkowski. Symmetry conditions of boolean functions in complex hadamard transform. *Electronic Letters*, 34:1634–1635, 1998.
- [RSA] RSAT. <http://reasoning.cs.ucla.edu/rsat/>.
- [Sab05] A. Sabharwal. Symchaff: A structure-aware satisfiability solver. In *American Association for Artificial Intelligence (AAAI)*, pages 467–474, Pittsburgh, PA, 2005.
- [SAT] SAT competition. <http://www.satcompetition.org>.
- [Sha38] C. E. Shannon. A symbolic analysis of relay and switching circuits. *Trans. AIEE*, 57:713–723, 1938.
- [Sha49] C. E. Shannon. The synthesis of two-terminal switching circuits. *Bell Systems Technical Journal*, 28(1):59–98, 1949.
- [Sle54] D. Slepian. On the number of symmetry types of boolean functions of n variables. *Canadian J. Math.*, 5:185–193, 1954.
- [TMS94] C.-C. Tsai and M. Marek-Sadowska. Detecting symmetric variables in boolean functions using generalized reed-muller forms. In *Internationa-*

- tional Conferece on Circuits and Systems*, volume 1, pages 287–290, 1994.
- [Urq87] A. Urquhart. Hard examples for resolution. *Journal of the Association for Computing Machinery*, 34(1):209–219, 1987.
- [U.S] U.S. census bureau.
http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html.
- [VB01] M. N. Velev and R. E. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. In *Proc. Design Automation Conference (DAC)*, pages 226–231, New Orleans, Louisiana, 2001.
- [You30] A. Young. On quantitative substitutional analysis. *Proc. London Math. Soc. (2)*, 31:273–288, 1930.
- [ZMBCJ06] J. S. Zhang, A. Mishchenko, R. Brayton, and M. Chrzanowska-Jeske. Symmetry detection for large boolean functions using circuit representation, simulation, and satisfiability. In *Design Automation Conference*, pages 510–515, 2006.

Chapter 14

Minimal Unsatisfiability and Autarkies

Hans Kleine Büning and Oliver Kullmann

The topic of this chapter is the study of certain forms of “redundancies” in propositional conjunctive normal forms and generalisations. In Sections 14.1 - 14.7 we study “minimally unsatisfiable conjunctive normal forms” (and generalisations), unsatisfiable formulas which are irredundant in a strong sense, and in Sections 14.8 - 14.13 we study “autarkies”, which represent a general framework for considering redundancies. Finally in Section 14.14 we collect some main open problems.

14.1. Introduction

A literal is a variable or a negated variable. Let X be a set of variables, then $\text{lit}(X)$ is the set of literals over the variables in X . Clauses are disjunctions of literals. Clauses are also considered as sets of literals. A propositional formula in conjunctive normal form (CNF) is a conjunction of clauses. CNF formulas will be considered as multi-sets of clauses. Thus, they may contain multiple occurrences of clauses. The set of all variables occurring in a formula φ is denoted as $\text{var}(\varphi)$. Next we introduce formally the notion of *minimal unsatisfiability*. Please note that in some of the older papers minimal unsatisfiable formulas are sometimes called “critical satisfiable”.

Definition 14.1.1. A set of clauses $\{f_1, \dots, f_n\} \in \text{CNF}$ is called *minimal unsatisfiable* if $\{f_1, \dots, f_n\}$ is unsatisfiable and for every clause f_i ($1 \leq i \leq n$) the formula $\{f_1, \dots, f_{i-1}, f_{i+1}, \dots, f_n\}$ is satisfiable. The set of minimal unsatisfiable formulas is denoted as MU.¹

The complexity class D^P is the set of problems which can be described as the difference of two NP-problems, i.e., a problem Z is in D^P iff $Z = X - Y$ where X and Y are in NP. The D^P completeness of MU has been shown by a reduction of the D^P -complete problem UNSAT-SAT [PW88]. UNSAT-SAT is the set of pairs (F, G) for which F is unsatisfiable and G is satisfiable.

¹Depending on the author, instead of “minimal unsatisfiable formulas” also “minimally unsatisfiable formulas” can be used (or is preferable).

Theorem 14.1.1. [PW88] MU is D^P -complete.

The idea of the proof is as follows. At first, we show that MU is in D^P . Let Y be the set of satisfiable CNF and let X be the set of CNF such that after removal of any clause they are satisfiable. Then we obtain $MU = X - Y$ and therefore MU is in D^P . For the hardness we assign to every pair of 3-CNF formulas (F, G) a formula $H(F, G)$, such that (F, G) is in UNSAT-SAT if and only if $H(F, G)$ is minimal unsatisfiable:

- Let $F = \{f_1, \dots, f_n\}$ be in 3-CNF with $f_i = (L_{i,1} \vee L_{i,2} \vee L_{i,3})$.
- For new variables x_1, \dots, x_n let $\pi_i = (x_1 \vee \dots \vee x_{i-1} \vee x_{i+1} \vee \dots \vee x_n)$.
- We define

$$H_1(F) = \bigwedge_{1 \leq i \leq n} (f_i \vee \pi_i) \wedge \bigwedge_{1 \leq i \leq n, 1 \leq j \leq 3} (\neg L_{i,j} \vee \pi_i \vee \neg x_i) \wedge \bigwedge_{1 \leq i < j \leq n} (\neg x_i \vee \neg x_j).$$

- Let $G = \{g_1, \dots, g_m\}$ be in 3-CNF with $g_i = (L'_{i,1} \vee L'_{i,2} \vee L'_{i,3})$.
- For new variables y_1, \dots, y_m let $\varphi_i = (y_1 \vee \dots \vee y_{i-1} \vee y_{i+1} \vee \dots \vee y_m)$.
- We define

$$H_2(G) = \bigwedge_{1 \leq i \leq n} (g_i \vee \varphi_i) \wedge \bigwedge_{1 \leq i \leq m, 1 \leq j \leq 3} (\neg L'_{i,j} \vee \varphi_i \vee \neg x_i) \wedge \bigwedge_{1 \leq i < j \leq m} (\neg y_i \vee \neg y_j) \wedge (y_1 \vee \dots \vee y_m).$$

- Then we have $H_2(G)$ is unsatisfiable. Moreover, F is unsatisfiable if and only if $H_1(F) \in MU$, and G is satisfiable if and only if $H_2(G) \in MU$.
- For $H_1(F) = \bigwedge_i \sigma_i$ and $H_2(G) = \bigwedge_j \tau_j$ we define $H(F, G) = \bigwedge_{i,j} (\sigma_i \vee \tau_j)$.
- Then (F, G) is in UNSAT-SAT if and only if $H(F, G)$ is in MU.

14.2. Deficiency

A very interesting measure for the complexity of formulas in CNF is the so-called “deficiency” which indicates the difference between the number of clauses and the number of variables.

Definition 14.2.1. Let F be a formula in CNF with n clauses and k variables. Then the *deficiency* of F is defined as $d(F) = n - k$. Further, we define the *maximal deficiency* as $d^*(F) = \max\{d(G) \mid G \subseteq F\}$.

Let k -CNF be the set of CNF-formulas with deficiency (exactly) k . Then the satisfiability problem for k -CNF is NP-complete. That can be shown easily by a reduction to SAT. Similarly, let $CNF^*(k)$ be the set of formulas with maximal deficiency k , i.e. $\{F \in CNF : d^*(F) = k\}$. In [FKS02] it has been shown that the satisfiability problem for these classes is solvable in polynomial time.

The set of minimal unsatisfiable formulas with deficiency k is denoted as $MU(k)$. It has been shown in [AL86] that any minimal unsatisfiable formula has a deficiency greater than 0, that means, every minimal unsatisfiable formula contains more clauses than variables. With respect to deficiency, one of the main results is the following

Theorem 14.2.1. [FKS02] For fixed integer k the problem $\text{MU}(k)$ is solvable in polynomial time.

Instead of asking whether a formula is in $\text{MU}(k)$ we may be interested in whether a formula contains a $\text{MU}(k)$ -subformula. This problem (i.e., the set of formulas) is denoted by $\text{sup-MU}(k)$ and is NP-complete [Sze01, KZ02a].

Formulas in conjunctive normal form can be represented as matrices (“variable-clause matrices”). The columns are the clauses and for each variable there is a row.

Example 14.2.1. For example, the formula $F = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2)$ can be written as

$$\begin{pmatrix} x_1 & x_1 & & \neg x_1 & \neg x_1 \\ x_2 & & \neg x_2 & x_2 & \neg x_2 \\ x_3 & \neg x_3 & x_3 & & \neg x_3 \end{pmatrix}$$

If the order of the variables is fixed then sometimes we replace the positive occurrence of a variable by the symbol ‘+’ and the negative occurrence by the symbol ‘-’. For the fixed order x_1, x_2, x_3 and the formula given above, we have

$$\begin{pmatrix} + & + & & - & - \\ + & & - & + & - \\ + & - & + & & - \end{pmatrix}.$$

The transposed of the variable-clause matrix is called the “clause-variable matrix”, and is studied further (as a matrix over $\{-1, 0, +1\}$) in [Kul03]; see Subsection 14.11.3.2 for further details on matrix representations of formulas.

14.2.1. Splitting

For a formula $F \in \text{MU}$ and a variable x we can assign the value true to x and reduce the formula by removing any clause with x and deleting any occurrence of $\neg x$. The formula we obtain is unsatisfiable and therefore contains at least one minimal unsatisfiable formula. If we perform the assignment also for $\neg x$ then we get two MU -formulas (a choice for each branch), and in this sense the assignment and the reduction splits the formula F into two MU -formulas. The splitting can be used to investigate the structure of MU -formulas and is a helpful tool for proofs; for an early reference see [KB99].

Definition 14.2.2. For $F \in \text{MU}$ and a variable x of F there exist formulas $B_x, C, B_{\neg x}$ in which neither x nor $\neg x$ occur such that

1. the formula F can be represented as

$$F = \{x \vee f_1, \dots, x \vee f_s\} \cup B_x \cup C \cup B_{\neg x} \cup \{\neg x \vee g_1, \dots, \neg x \vee g_t\};$$

2. $\{f_1, \dots, f_s\} \cup B_x \cup C$, denoted by F_x , is in MU , and $\{g_1, \dots, g_t\} \cup B_{\neg x} \cup C$, denoted by $F_{\neg x}$, is in MU . F_x and $F_{\neg x}$ may consist of the empty clause only.

We call $(F_x, F_{\neg x})$ a *splitting* of F on x .

Example 14.2.2. Continuing Example 14.2.1, we obtain the splitting

$$\begin{aligned} F_{\neg x_1} &= (x_2 \vee x_3) \wedge (\neg x_3) \wedge (\neg x_2 \vee x_3) \\ F_{x_1} &= (\neg x_2 \vee x_3) \wedge (x_2) \wedge (\neg x_2 \vee \neg x_3). \end{aligned}$$

The corresponding matrix representations are

$$\begin{pmatrix} x_2 & \neg x_2 \\ x_3 & \neg x_3 & x_3 \end{pmatrix}, \begin{pmatrix} \neg x_2 & x_2 & \neg x_2 \\ x_3 & & \neg x_3 \end{pmatrix}.$$

For all k , all $F \in \text{MU}(k)$, all variables x in F and all splittings $(F_x, F_{\neg x})$ of F on x we have:

1. $d(F_x), d(F_{\neg x}) \leq k$ (see Corollary 7.10 in [Kul03], considering the larger class of “matching lean” clause-sets as discussed later in Subsection 14.11.2).
2. If all variables occur positively as well as negatively at least twice, then $d(F_x), d(F_{\neg x}) < k$ (Theorem 2 in [KB00]; see Lemma 3.9 in [Kul00a] for a somewhat stronger statement, also considering the larger class of “lean” clause-sets as discussed later in Subsection 14.8.3).
3. If $k \geq 2$, then (by Lemma 3.10 in [Kul00a]) there is some variable x' such that for some splitting $(F_{x'}, F_{\neg x'})$ we have $d(F_{x'}) < k$ and $d(F_{\neg x'}) < k$. Such a variable can be computed in polynomial time.

That by splitting the deficiency can be reduced has to do with the “expansion” of variables, and is discussed further in Section 4.4 in [Kul07a]. If the common part of both formulas, C , is empty, we call the splitting *disjunctive*. We say a formula has a *unique splitting* for a variable x , if there exists exactly one pair of splitting formulas $(F_x, F_{\neg x})$. In general, splittings are not unique. If a MU-formula F has a disjunctive splitting on x then however the disjunctive splitting on x is unique. See Section 3 in [KZ03] for further investigations into splittings.

14.2.2. Structure of MU(1) and MU(2)

A class of MU-formulas with deficiency 1 is the set Horn-MU of minimal unsatisfiable Horn formulas. That together with the linear-time decidability can easily be shown by an induction on the number of variables. But MU(1) contains more formulas. The structure of MU(1)-formulas is well understood. Every MU(1)-formula can be represented as a *basic matrix* and vice versa [DDKB98]. Basic matrices are defined inductively as follows:

1. $(+ \ -)$ is a basic matrix.
2. If B is a basic matrix then

$$\begin{pmatrix} B & 0 \\ b_1 & + \end{pmatrix}, \begin{pmatrix} B & 0 \\ b_2 & - \end{pmatrix}$$

are basic matrices, where b_1, b_2 are non-null row vectors without $+$ resp. without $-$.

3. If B_1 and B_2 are basic matrices and b_1, b_2 are as above, then

$$\begin{pmatrix} B_1 & 0 \\ b_1 & b_2 \\ 0 & B_2 \end{pmatrix}$$

is a basic matrix.

Here, “+” represents a positive literal, and “−” a negative literal. We usually omit zeros and write them as blank fields. The rows belong to the variables and the columns to the clauses.

Example 14.2.3. The formula $x \wedge \neg x$ has the basic matrix $(+ -)$. For the formula $(x_1 \vee x_2) \wedge (\neg x_1) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$ we have the basic matrix

$$\begin{pmatrix} + & - & & \\ + & & - & - \\ & & + & - \end{pmatrix}.$$

Since basic matrices have a disjunctive splitting, we can always find a disjunctive splitting for MU(1)-formulas. Moreover, it is easy to see that any minimal unsatisfiable Horn formula is in MU(1), whereas for 2-CNF minimal unsatisfiable formulas with an arbitrary deficiency can be constructed. For an approach based on a tree representation of the elements of MU(1) see [Kul00a]. We remark that the class MU(1) has been studied rather extensively from the point of view of “qualitative matrix analysis”; see Subsection 14.12.1 for some general introduction.

With respect to the structure of the matrices for greater deficiencies only very few results are known. For example, every MU(2)-formula in which each literal occurs at least twice has the following form except for renaming [KB00]:

$$\begin{pmatrix} x_1 & \neg x_1 & \neg x_2 & \cdots & \neg x_{n-1} & \neg x_n & \neg x_1 \\ x_2 & x_2 & x_3 & \cdots & x_n & x_1 & \neg x_2 \\ x_3 & & & & & & \neg x_3 \\ \vdots & & & & & & \vdots \\ x_n & & & & & & \neg x_n \end{pmatrix}$$

See [SD97] for some initial study on MU(3) and MU(4).

14.3. Resolution and Homomorphism

It is well-known that in the worst-case resolution refutations of unsatisfiable formula require super-polynomially many resolution steps. For fixed deficiency, for each $F \in \text{MU}(k)$ there is a resolution refutation with not more than $2^{k-1}n$ resolution steps. The proof follows immediately by splitting property 3 from Section 14.2.1 together with the characterisation of MU(1) from Subsection 14.2.2.

It is easy to see that MU(k) is closed under $(1, *)$ -resolution (also called *singular DP-resolution*) for every k . That means, if $F = \{L \vee f, \neg L \vee g_1, \neg L \vee g_2, \dots, \neg L \vee g_s\} + F'$ \in MU(k) and L as well as $\neg L$ do not occur in F' , then

$\{f \vee g_1, f \vee g_2, \dots, f \vee g_s\} + F' \in \text{MU}(k)$. Please note that the literal L occurs only once in the formula, and that we simultaneously resolve on all clauses with literal L . In that sense, $(1, *)$ -resolution is a restricted version of hyperresolution.

We say a formula is *read-once refutable* if the formula has a resolution refutation tree in which each input clause is used at most once, and such a refutation is called a *read-once refutation*. ROR is the class of read-once refutable formulas and known to be NP-complete [IM95]. The following example is a formula for which a read-once refutation exists, no proper subformula has a read-once refutation, but the formula is not minimal unsatisfiable.

$$\left(\begin{array}{cccccccc} z & u & \neg x & \neg x & \neg a & \neg a & a & y & \neg z & \neg u \\ & & \neg y & \neg b & \neg y & \neg b & x & b & y & a \\ & & & & & & & & b & \neg x \end{array} \right).$$

Let $\text{ROR-MU}(k)$ be the set of minimal unsatisfiable formulas with deficiency k for which a read-once refutation exists. $\text{ROR-MU}(k)$ is decidable in polynomial time for any fixed k [KZ02a]. For the class $\text{Sup-ROR-MU}(k) := \{F \mid \exists G \in \text{ROR-MU}(k) : G \subseteq F\}$ it has been shown in [KZ02a, Sze01] that the decision problem is NP-complete. In [FSW06] it has been shown that the *Short Resolution Refutation* (SRR) and the *Small Unsatisfiability Subset* (SUS) problems are likely not fixed-parameter tractable, however the restrictions to planar formulas or to formulas with fixed bounds on clause-length and variable-occurrence are fixed-parameter tractable. Here SRR is the problem whether for a parameter t a refutation with at most t steps exists, and SUS is the problem of deciding whether a formula contains an unsatisfiable sub-formula with at most t clauses.

In [Sze01] homomorphisms for CNF-formulas have been introduced as a tool for proving the unsatisfiability of formulas. For CNF-formulas H and F , a mapping $\phi : \text{lit}(H) \rightarrow \text{lit}(F)$ is called a *homomorphism* from H to F if $\phi(\neg x) = \neg\phi(x)$ for every variable x , and $\phi(H) = \{\phi(h) \mid h \in H\} \subseteq F$, where $\phi(h) = \{\phi(L_1), \dots, \phi(L_m)\}$ for a clause $h = \{L_1, \dots, L_m\}$. According to [Sze01], for every tree resolution proof T one can find in polynomial time a formula $H \in \text{MU}(1)$ and a homomorphism $\phi : H \rightarrow \text{pre}(T)$, where $\text{pre}(T)$ is the set of all premises of T (i.e., the clauses labelling the leaves) such that $\phi(H) = \text{pre}(T)$, and $|H|$ equals the number of leaves of T .

Let \mathcal{C} be a class of CNF-formulas. We define $\mathcal{HOM}(\mathcal{C})$ as the set of pairs (F, G) of CNF-formulas such that a homomorphism from F to G exists with image all of G . The problem $\mathcal{HOM}(\text{Horn-MU})$ is NP-complete. Clearly, the problem is in NP. The completeness can be shown by a reduction to the 3-colouring problem for graphs. Furthermore, for fixed $k \geq 1$ the problem $\mathcal{HOM}(\text{MU}(k))$ is NP-complete, even we allow another fixed parameter $t \geq 1$ and consider all pairs (F, G) with $F \in \text{MU}(k)$ and $G \in \text{MU}(t)$ [KX05].

We say (H, ϕ) is a *representation* of F if $\phi : H \rightarrow F$ is a homomorphism with $\phi(H) = F$; so $\mathcal{HOM}(\mathcal{C})$ is the set of all pairs (F, G) such that F can be made a representation of G . Let (H, ϕ) be a representation of F , where F and H are minimal unsatisfiable formulas. If the shortest resolution refutation of F requires m steps then the shortest refutation for H needs at least m steps [KZ02b]. For all $k, t \geq 1$ and for all $F \in \text{MU}(k)$ a representation of F by a $\text{MU}(t)$ -formula can be computed in polynomial time.

A homomorphism ϕ with $\phi(H) = F$ is termed *clause-preserving* for H and F if H and F have the same number of clauses. As a consequence of results shown in [Sze01] and [KZ02b], we have that for every formula F for which a disjunctive splitting tree exists, we can find a formula $H \in \text{MU}(1)$ for which a clause-preserving homomorphism ϕ with $\phi(H) = F$ exists, and vice versa.

14.4. Special Classes

In this section we present some MU-problems for restricted classes of formulas and some additional constraints.

14.4.1. 2-CNF-MU

For every $k \geq 1$ there is a formula $F \in \text{2-CNF-MU}(k)$, the set of minimal unsatisfiable formulas with deficiency k consisting of 2-clauses. Whether a 2-CNF-formula is in MU can be decided in linear time. In a 2-CNF-MU formula, each literal occurs at most twice. Every 2-CNF-MU formula can be reduced by iterative (1,*)-resolution in linear time to a 2-CNF-MU formula in which each literal occurs exactly twice. These formulas have the following structure except for renaming

$$\begin{pmatrix} x_1 \neg x_2 \cdots \neg x_{n-1} \neg x_n \neg x_1 & x_2 \cdots x_{n-1} & x_n \\ x_2 & x_3 \cdots x_n & x_1 \neg x_2 \neg x_3 \cdots \neg x_n \neg x_1 \end{pmatrix}.$$

14.4.2. Hitting Formulas

Let $\text{HIT} := \{F \in \text{CNF} \mid \forall f, g \in F, f \neq g \exists L \in f : \neg L \in g\}$ denote the set of “hitting formulas”, formulas in which every pair of clauses has a complementary pair of literals. Let HIT-MU be the set of minimal unsatisfiable hitting formulas. In [Iwa89], a satisfiability algorithm has been introduced which counts for a given formula the number $\phi(F)$ of falsifying truth assignments (total assignments which for at least one clause falsify every literal in it). If the number is $\phi(F) = 2^n$, where n is the number of variables, then the formula is unsatisfiable, and otherwise satisfiable. The idea of this counting is based on “independent” sets of clauses. A set of clauses S is termed *independent* if every pair of (different) clauses contains no complementary pair of literals. The calculation of $\phi(F)$ can be expressed as follows for a formula $F = \{f_1, \dots, f_m\}$:

$$\phi(F) = \sum_{1 \leq i \leq m} \sum_{S \in \text{IND}_i(F)} (-1)^{i-1} \cdot 2^{n-|\text{var}(S)|}$$

where $\text{IND}_i(F) := \{S \subseteq F \mid S \text{ independent and } |S| = i\}$. F is a hitting formula if and only if $\text{IND}_i(F) = \emptyset$ for $i \geq 2$; thus for $F \in \text{HIT}$ we have

$$\phi(F) = \sum_{f \in F} 2^{n-|f|}.$$

It follows for F in HIT:

1. $\sum_{f \in F} 2^{-|f|} \leq 1$;
2. $\sum_{f \in F} 2^{-|f|} < 1 \iff F \in \text{SAT}$;
3. $\sum_{f \in F} 2^{-|f|} = 1 \iff F \in \text{MU}$.

A special class of hitting clause-sets are k -regular hitting clause-sets, where between two different clauses always *exactly* k complementary (or “clashing”) pairs of literals exist. A k -regular hitting clause-set for $k \neq 1$ is satisfiable, due to the completeness of resolution. The unsatisfiable 1-regular hitting clause-sets have been characterised in [Kul04a] as those elements of $\text{MU}(1)$ which are “saturated” or “maximal” (see below); an alternative combinatorial proof (not using methods from linear algebra as in [Kul04a]) has been given in [SST07].

14.4.3. Stronger forms of minimal unsatisfiability

Let $A \leq_p B$ denote the polynomial reducibility between A and B (i.e., A can be polynomially reduced to B), while $A =_p B$ is an abbreviation for $A \leq_p B$ and $B \leq_p A$. It is known that MARG-MU , MAX-MU and Almost-Unique-MU are D^P -complete, while $\text{Unique-MU} =_p \text{Unique-SAT} \leq_p \text{Dis-MU}$, where these classes are defined as follows:

1. *Maximal formulas* MAX-MU : A clause f of an MU -formula F is called *maximal* in F if for any literal L occurring neither positively nor negatively in f the formula obtained from F by adding L to f is satisfiable. Then $F \in \text{MAX-MU}$ if f is maximal in F for any $f \in F$. Another notion used in the literature is “saturated clause-sets”
2. *Marginal formulas* MARG-MU : A formula $F \in \text{MU}$ is called *marginal* w.r.t. a literal L if removing an arbitrary occurrence of L from F produces a non-minimal unsatisfiable formula. We say F is *marginal* if F is marginal w.r.t. all literals occurring in F . Or in other words, a formula is marginal if and only if removing an arbitrary literal always leads to a non-minimal unsatisfiable formula. The set of marginal formulas is denoted as MARG-MU .
3. *Unique minimal unsatisfiable formulas* Unique-MU : Based on the well-known concept of Unique-SAT , the set of formulas having exactly one satisfying truth assignment, we define the class Unique-MU as the class of MU -formulas F for which for any clause $f \in F$ we have $F - \{f\} \in \text{Unique-SAT}$.
4. *Almost unique formulas* Almost-Unique-MU : A weaker notion which demands that except for one clause f , the reduced formula $F - \{f\}$ must be in Unique-SAT .
5. *Disjunctive Formulas* Dis-MU : We define $F \in \text{Dis-MU}$ if $F \in \text{MU}$ and for any $x \in \text{var}(F)$ every splitting of F on x is disjunctive. Please note, that in case of a disjunctive splitting on a variable x the splitting on x is unique.

See [KZ07a] for more information on these classes.

14.4.4. Regarding the number of literal occurrences

For fixed $k \geq 3$, the problem “ k -CNF-MU”, recognising minimal unsatisfiable formulas in k -CNF, remains D^P -complete. The D^P -completeness of k -CNF-MU follows from the D^P -completeness of MU. We only have to replace longer clauses with shorter clauses by introducing new variables and by splitting the long clauses. Let k -CNF p be the class of k -CNF-formulas in which each literal occurs at least p times. It has been shown in [KBZ02] that 3-CNF-MU 3 , and for any fixed $k \geq 4$ and any $p \geq 2$ the classes k -CNF-MU p are D^P -complete. But whether a formula in 3-CNF-MU exists in which each literal in F occurs at least 5 times is not known. More results can be found for example in [KZ02a].

14.4.5. Complement-invariant formulas

A formula F is *complement-invariant* if for every clause $\{L_1, \dots, L_k\} \in F$ also the complemented clause is in F , that is $\{\neg L_1, \dots, \neg L_k\} \in F$. A complement-invariant F , which additionally consists only of positive and negative clauses, can be identified with the underlying variable-hypergraph $\mathfrak{H}(F)$, whose vertices are the variables of F while the hyperedges are the variable-sets of clauses of F . It is easy to see that such F is satisfiable if and only if $\mathfrak{H}(F)$ is 2-colourable (i.e., vertices can be coloured using two colours such that no hyperedge is monochromatic), while F is minimally unsatisfiable if and only if $\mathfrak{H}(F)$ is minimally non-2-colourable (or “critically 3-colourable”), i.e., G is not 2-colourable but removing any hyperedge renders it 2-colourable. In this way critical-3-colourability of hypergraphs is embedded into the study of minimal unsatisfiable formulas. Complement-invariant formulas have been studied at various places, but apparently the first systematic treatment is in [Kul07b]. The deficiency of complement-invariant formulas F is less useful here, but the *reduced deficiency* $d_r(F) := d(\mathfrak{H}(F))$ is the central notion, where the deficiency $d(G)$ of a hypergraph is the difference of the number of hyperedges and the number of vertices. As shown in [Sey74] (translated to our language), the reduced deficiency of minimally unsatisfiable $F(G)$ is at least 0. Solving a long outstanding open question, in [RST99, McC04] it was shown (again, in our language) that the decision problem whether a complement-invariant formula is minimally unsatisfiable with reduced deficiency 0 is decidable in polynomial time; more on the strong connections to the satisfiability problem the reader finds in [Kul07b], while in Subsection 14.12.2 of this chapter we further discuss this problem from the point of view of autarky theory.

14.4.6. Weaker forms of minimal unsatisfiability

The question whether an arbitrarily given formula can be transformed into a minimal unsatisfiable formula by replacing some literals with their complements has been studied in [Sze05]. It has been shown in [Sze05] that the problem whether a formula is satisfiable and remains satisfiable under such replacements is Π_2^P -complete.

The class of formulas which are the union of minimally unsatisfiable formulas has been studied at different places; see for example [KLMS06, Kul07a]. A generalisation of minimal unsatisfiability based on autarkies is discussed in Subsection 14.4.7.

14.4.7. Generalising minimal unsatisfiability for satisfiable formulas

Similarly to minimal unsatisfiable formulas, where removing a clause leads to a satisfiable formula, we say a formula F is *clause-minimal* (or “irredundant”) if removing a clause f from F results in a formula not equivalent to F , i.e. $F - \{f\} \not\equiv F$. Let $\text{CL-MIN} := \{F \in \text{CNF} \mid \forall f \in F : F - \{f\} \not\equiv F\}$ and $\text{CL-MIN}(k)$ the set of CL-MIN-formulas with deficiency k . The problems CL-MIN and $\text{SAT} \cap \text{CL-MIN}$ are known to be NP-complete [PW88]. Furthermore, we have $\text{MU} = \text{UNSAT} \cap \text{CL-MIN}$, and for any fixed k the problem $\text{CL-MIN}(k)$ is NP-complete [KZ05]. Further results one finds in [Kul07a].

14.4.8. Extending minimal unsatisfiable formulas

The extension problem is the problem of determining whether for pairs of formulas (F, H) there is a formula G for which $F + G \equiv H$ and $\text{var}(G) \subseteq \text{var}(H + F)$. Without any restriction this question is equivalent to whether $H \models F$ (that is, whether H implies F). The latter problem is known to be coNP-complete. Now we are looking for extensions for MU-formulas. Let $\text{MU-EXT} = \{F \mid \exists G : F + G \in \text{MU} \text{ and } \text{var}(G) \subseteq \text{var}(F)\}$. Then we have $\text{MU-EXT} = \text{CL-MIN}$. Suppose, a CNF-formula F can be extended to a minimal unsatisfiable formula. Then it is easy to see that F is clause minimal. For the inverse direction suppose $F \in \text{CL-MIN}$ is satisfiable. Let $\{t_1, \dots, t_r\}$ be the set of satisfying (total) truth assignments for F and let G consist of the r clauses exactly falsified by the assignments t_i ; then $F + G$ is in MU. More details and further classes can be found in [KZ05]

14.5. Extension to non-clausal formulas

In this section, we will extend the notion of “minimal unsatisfiability” and “deficiency” to non-clausal propositional formulas, where for the sake of a simplified representation we only allow the operations “binary and”, “binary or” and “not”. Furthermore we study only formulas in “negation normal form”, where we have negations only at the leaves, i.e., we study binary nested and’s and or’s of literals. The length of a propositional formula is the number of occurrences of literals. More formally, the length can be defined as $\ell(L) = 1$ for a literal L and $\ell(F \wedge G) = \ell(F \vee G) = \ell(F) + \ell(G)$. The number of occurrences of \wedge -symbols in a formula F is denoted by $\#\wedge(F)$, whereas $\#\vee(F)$ is the number of \vee -symbols. Obviously, we have $\ell(F) = 1 + \#\wedge(F) + \#\vee(F)$ (since the operations are binary). Following [KZ07b], we define:

Definition 14.5.1. Let F be a propositional formula in negation normal form with $n(F)$ variables. The *cohesion* $D(F)$ is defined as $D(F) = 1 + \#\wedge(F) - n(F)$.

Example 14.5.1. The formula $F = (x \wedge y) \vee (\neg x \wedge (z \vee \neg y))$ has the cohesion $D(F) = 1 + \#\wedge(F) - n(F) = 1 + 2 - 3 = 0$. An application of the distributive law $F \wedge (G \vee H) \equiv (F \wedge G) \vee (F \wedge H)$ may change the cohesion. For example, from $F = z \wedge (x \vee y)$ we obtain the formula $F' = (z \wedge x) \vee (z \wedge y)$. The cohesion of F is -1 and the cohesion of F' is 0 .

For propositional formulas in negation normal form the so-called ‘‘Tseitin procedure’’ generates satisfiability-equivalent formulas in CNF. Two formulas F and G are satisfiability-equivalent, if F is satisfiable if and only if G is satisfiable. For F a propositional formula in negation normal form, the Tseitin procedure replaces step by step subformulas of the form $(F_1 \wedge F_2) \vee F_3$ by the formula $(z \vee F_1) \wedge (z \vee F_2) \wedge (\neg z \vee F_3)$ for a new variable z . Let $\text{ts}(F)$ denote a formula in CNF obtained from F by this Tseitin procedure, eliminating the non-determinism in the procedure in some (arbitrary) way. Because the Tseitin procedure adds as many \wedge -symbols as new variables, we obtain $D(\text{ts}(F)) = D(F)$.

Lemma 14.5.1. *Let F be a propositional formula in negation normal form. Then we have $D(F) = \ell(F) - \#\vee(F) - n(F)$. and if $F \in \text{CNF}$, then $D(F) = d(F)$.*

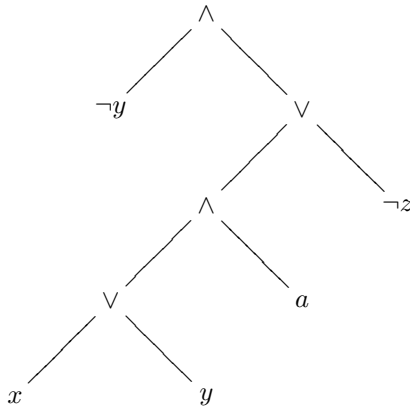
Minimal unsatisfiability is defined for formulas in conjunctive normal form. The formulas are unsatisfiable, and after eliminating an arbitrary clause the formulas are satisfiable. In non-clausal formulas, instead of the deletion of a clause we will remove so-called ‘‘or-subformulas’’ based on a representation of formulas as trees:

- If the formula is a literal L , the associated tree T_L is a node labelled with L .
- For a formula $F \wedge G$ (resp. $F \vee G$) let T_F and T_G be the associated trees. Then we have a root labelled with \wedge (resp. \vee) and the subtrees T_F and T_G . That means, the successor nodes of the \wedge -node (resp. \vee -node) are the roots of T_F and T_G .

Please notice that the leaves of the tree are labelled with the literals of the formula. Let T be the associated tree of a formula. An *or-subtree* of T is a subtree T' , whose root is either an \vee -node or a literal which is successor of an \wedge -node. An or-subtree T' of T is again a representation of a propositional formula, say $F_{T'}$, which is a subformula of the propositional formula of F . The formula $F_{T'}$ is called an *or-subformula* of F . In case of formulas in conjunctive normal form, the or-subformulas are exactly the clauses.

Example 14.5.2. For the formula $(\neg y) \wedge ((x \vee y) \wedge a) \vee \neg z$ the associated tree

is



and the four or-subformulas are $(\neg y)$, $((x \vee y) \wedge a) \vee (\neg z)$, $(x \vee y)$, and a .

In case of a propositional formula F (as always in negation normal form), an or-subformula of F is a subformula of F . For formulas in conjunctive normal form, that is a conjunction of clauses, a satisfiable formula remains satisfiable after the deletion of any clause. Similarly, after the deletion of an or-subtree in a satisfiable formula the formula is still satisfiable. That follows from the definition of or-subtrees, because the predecessor node of an or-subtree is an \wedge -node, and so the deletion of an or-subtree does not decrease the truth-value of the formula at that \wedge -node, while the formulas are in negation normal form and thus all operations are monotonically increasing.

Definition 14.5.2. A propositional formula F in negation normal form is in MU^* if F is unsatisfiable and eliminating an arbitrary or-subformula yields a satisfiable formula. The set of formulas in MU^* with cohesion (exactly) k is denoted by $MU^*(k)$.

It can easily be shown that for a CNF-formula being minimally unsatisfiable and being element of MU^* are equivalent properties. By the property of the Tseitin procedure, and the results known for MU and $MU(k)$ the following theorem holds [KZ07b].

Theorem 14.5.2. *The following holds for the notion of minimal unsatisfiable propositional formulas according to Definition 14.5.2:*

1. MU^* is D^P -complete.
2. For $F \in MU^*$ we have $D(F) \geq 1$.
3. For fixed k , $MU^*(k)$ is decidable in polynomial time.

There is an alternative definition of the class MU^* , based on the complementation of literal occurrences. Let MU_L be the set of unsatisfiable formulas F such that for all literal occurrences of F its replacement by its complement renders the formula satisfiable. Obviously, for formulas in CNF, any formula in MU_L is in MU and vice versa. For the general case, $MU^* = MU_L$ can be shown by an analysis of the Tseitin procedure $ts(F)$.

14.6. Minimal Falsity for QBF

The concept of minimal unsatisfiability for CNF can be extended to QCNF, the class of closed quantified Boolean formulas in prenex form and with matrix in CNF; see Chapter 7 in [KBL99] for a general introduction to quantified propositional logic. All formulas F in QCNF have the form $F = Q_1x_1 \cdots Q_nx_nf$, where $Q_i \in \{\exists, \forall\}$ and f is a propositional formula in conjunctive normal form using (only) the variables x_1, \dots, x_n . $Q_1x_1 \cdots Q_nx_n$ is the *prefix* of F , and f is called the *matrix* of F . The set of existential variables is denoted by $\text{var}_{\exists}(F)$, while $\text{var}(F)$ is the set of all variables. Let $F = Q_1x_1 \cdots Q_nx_nf$, $F' = Q_1x_1 \cdots Q_nx_nf'$ be two QCNF formulas with the same prefix. We say that F' is a *subformula* of F , denoted by $F' \subseteq F$, if f' is a subformula of f . Let Qf be a formula in QCNF with prefix Q . Then $f_{|\exists}$ is the conjunction of clauses we obtain after the deletion of all occurrences of literals with universal variables from f . Please note, that the propositional formula $f_{|\exists}$ may contain multiple occurrence of clauses. A formula $Q(f_1 \wedge \cdots \wedge f_n)$ in QCNF is called *minimal false*, if the formula is false and for any clause f_i the formula $Q(f_1 \wedge \cdots \wedge f_{i-1} \wedge f_{i+1} \wedge \cdots \wedge f_n)$ is true. The class of minimal false formulas is denoted as MF.

The minimal falsity problem MF is PSPACE-complete [KZ06]. That can easily be shown by means of the PSPACE-completeness of the satisfiability problem for QCNF. Since the satisfiability problem for quantified Boolean Horn formulas (QHORN) and formulas with matrix in 2-CNF (Q2-CNF) is solvable in polynomial time, the minimal falsity problems for both classes are solvable in polynomial time, too. The notion of deficiency and maximal deficiency can be extended to QCNF, taking only the existential variables into account.

Definition 14.6.1. [KZ06]

1. Let $F = Qf \in \text{QCNF}$ with m clauses. The deficiency is defined as $d(F) = m - |\text{var}_{\exists}(f)|$.
2. The maximal deficiency of F is defined as $d^*(F) := \max\{d(F') : F' \subseteq F\}$.
3. Let k be fixed. The set of minimal false formulas with deficiency k is defined as $\text{MF}(k) = \{F : F \in \text{MF} \text{ and } d(F) = k\}$.

Some of the methods and techniques well-known for $\text{MU}(k)$ can be adapted in order to prove similar results. For example, a formula F is termed *stable* if for any proper subformula $F' \subset F$, $d(F') < d(F)$. The notion of stable QCNF formulas can be understood as a generalisation of matching lean CNF formulas [Kul03]. A formula $F \in \text{CNF}$ is matching lean if and only if $d(F') < d(F)$ for each proper subformula $F' \subset F$; see Subsection 14.11.2. By Definition 14.6.1, F is stable if and only if $F_{|\exists}$ is matching lean. Corollary 7.12 in [Kul03] says that the largest matching lean subformula can be computed in polynomial time. This implies that the largest stable subformula can also be computed in polynomial time.

Theorem 14.6.1. [KZ08a] *Let $F = Qf$ be a formula in QCNF.*

1. *If $f_{|\exists}$ is satisfiable or $d^*(F) \leq 0$, then F is true.*
2. *If $F \in \text{MF}$, then F is stable.*
3. *Any minimal false formula has deficiency strictly greater than 0.*

Besides MF(1), which is solvable in polynomial time as shown later in this section, the computational complexity of MF(k) is open, where only some non-trivial upper bounds are known.

Theorem 14.6.2. [KZ06] *Let k be fixed.*

1. *The satisfiability problem for QCNF with maximal deficiency k is in NP.*
2. *The minimal falsity problem MF(k) for QCNF is in D^P .*

The proof of Theorem 14.6.2 makes use of the size of models of satisfiable formulas in QCNF with maximal deficiency k . Let $F = \forall x_1 \exists y_1 \forall x_2 \exists y_2 \cdots \forall x_k \exists y_k f$ be a formula in QCNF. The formula is true if and only if there are Boolean functions (formulas) $\rho_i : \{0, 1\}^i \rightarrow \{0, 1\}$ for $i \in \{1, \dots, k\}$ (depending on x_1, \dots, x_i) such that

$$\forall x_1 \cdots \forall x_k f[y_1/\rho_1(x_1), \dots, y_k/\rho_k(x_1, \dots, x_k)]$$

is true. $f[y_1/\rho_1, \dots, y_k/\rho_k]$ denotes the formula obtained by simultaneously replacing in the matrix the occurrences of the existential variables y_i by the formulas ρ_i . In general a sequence of Boolean functions $M = (\rho_1, \dots, \rho_k)$ is called a *model* for F if the formula is true for these functions. We assume that the Boolean functions ρ_i are represented as propositional formulas. If the Boolean functions ρ_i are given as CNF-formulas, then we call M a *CNF-model* for F .

Theorem 14.6.3. [KZ08a] *For any $k \geq 1$ and any true QCNF formula $F = Qf$ with maximal deficiency $d^*(F) = k$, there is a set U with at most $2^{4k/3}$ universal variables such that F has a CNF-model $M = (\rho_1, \dots, \rho_m)$ where the formulas ρ_i are constants or fulfil $\text{var}(\rho_i) \subseteq U$ and have at most 2^k clauses.*

The size of the model functions only depends on k . By guessing a set U of size less than or equal to $2^{4k/3}$ and model formulas ρ_i , and replacing x_i by ρ_i , we obtain a universally quantified formula with universal variables in U . Since k is fixed, the satisfiability of these formulas can thus be decided in polynomial time.

QEHORN (QE2-CNF respectively) is the set of formulas in QCNF with $F = Qf$ for which $f_{|\exists}$ is a Horn formula (2-CNF formula respectively). That means, the existential part of the matrix is in HORN (2-CNF respectively). The satisfiability problem for both classes is PSPACE-complete. Through unfolding these formulas by setting the universal variables to true and false, we get an existentially quantified Horn formula (2-CNF formula respectively). Since any true QCNF formula with maximal deficiency $k \geq 0$ has a model over at most $2^{4k/3}$ universal variables, the length of the unfolded formula can be bounded polynomially in the length of the initial formula. Since the satisfiability of Horn and 2-CNF formulas is decidable in polynomial time, we obtain the following result.

Lemma 14.6.4. *Let k be fixed.*

1. *The satisfiability problem for formulas in QEHORN and QE2-CNF with maximal deficiency k is solvable in polynomial time.*
2. *The minimal falsity problem for formulas in QEHORN and QE2-CNF with deficiency k can be decided in polynomial time.*

Formulas in MF(1) are closely related to formulas in MU(1). Every formula $F = Qf$ in MF(1) has a matrix $f_{|\exists} \in \text{MU}(1)$. But the other direction does not hold. A simple counterexample is as follows.

Example 14.6.1. Consider

$$F = \forall x \exists y (\neg x \vee y) \wedge (x \vee \neg y).$$

$((\neg x \vee y) \wedge (x \vee \neg y))|_{\exists} = (\neg x \wedge x) \in \text{MF}(1)$, but the formula F is satisfiable (by the model $y = x$).

Theorem 14.6.5. [KZ08a] *The minimal falsity problem $\text{MF}(1)$ is solvable in polynomial time.*

The polynomial-time algorithm is based on the following observations:

1. The satisfiability problem for QCNF formulas with maximal deficiency 1 is solvable in polynomial time.
2. For any true QCNF formula with $d^*(F) = 1$, there exists at most one universal variable y , such that there is a model in which the model functions are either the constants 0 or 1, or the formulas y or $\neg y$.
3. The clauses of a $\text{MF}(1)$ -formula must satisfy some properties on the connectivity of clauses.

14.7. Applications and Experimental Results

The core problems of relevant applications in this area is to extract minimal unsatisfiable sub-formulas (“MUS’s”) of a CNF-formula, where more precisely three levels can be distinguished:

1. finding some “small” unsatisfiable sub-formula;
2. finding some minimal unsatisfiable sub-formula (MUS);
3. finding a smallest minimal unsatisfiable sub-formula.

Additionally we are also interested in finding all MUS’s. Regarding finding “small” unsatisfiable sub-formulas, [BS01, Bru03] enhance and modify DPLL-solvers so that they target “hard clauses”, from which the unsatisfiable sub-formula is obtained. [ZM04] also only finds “small” unsatisfiable sub-formulas, but uses the resolution refutation found by conflict-driven SAT solvers. Regarding finding MUS’s, a natural and simple algorithm is given in [vW08], while learning of conflict-driven SAT solvers is exploited in [OMA⁺04], and [GMP06, GMP07a] use the information obtained by local search solvers on maximally satisfiable sub-formulas (see [GMP07b] for generalisations regarding constraint satisfaction problems). For finding smallest MUS’s, in [Bru05] ideas from linear programming are applied to special classes of CNF-formulas, while a general branch-and-bound algorithm is given in [MLA⁺05]. Regarding finding all MUS’s, [LS05] exploit the duality between MUS’s and maximally satisfiable sub-formulas (“MSS’s”; the generalisation for non-boolean variables and irredundant clause-sets is given in Lemma 6.11 in [Kul07a]). Finally, connecting MUS’s with “autarkies” (as a weaker form of redundancies), we mention [KLMS06, LS08].

14.8. Generalising satisfying assignments through “autarkies”

At several places did we already encounter the notion of an “autarky”. In the subsequent sections we give an overview on the emerging theory of autarkies.

14.8.1. Notions and notations

In the previous section the “logical aspect” of SAT has been emphasised, and thus the basic objects of study were called “formulas”. Autarky theory takes a more combinatorial turn and needs to take full account of “syntactical details”, and thus we will now speak of *clause-sets*, which are (here only finite) sets of *clauses*, where a clause is a finite and complement-free set of literals. Some useful notions:

- The *empty clause* is denoted by $\perp := \emptyset$, while the *empty clause-set* is denoted by $\top := \emptyset$.
- For a literal x we denote the underlying variable by $\text{var}(x)$, while for a clause C we define $\text{var}(C) := \{\text{var}(x) : x \in C\}$, and $\text{var}(F) := \bigcup_{C \in F} \text{var}(C)$ for a clause-set F .
- The number of clauses of a clause-set F is $c(F) := |F| \in \mathbb{N}_0$, while the number of variables is $n(F) := |\text{var}(F)| \in \mathbb{N}_0$.

An important step is to emancipate the notion of a *partial assignment*, which is a map $\varphi : V \rightarrow \{0, 1\}$ where V is some finite set of variables (possibly the empty set); we use $\text{var}(\varphi) := V$ to denote the domain of φ .² Using $\varphi(\bar{v}) = \varphi(v)$, partial assignments are extended to literals (over their domain). The application of a partial assignment φ to a clause-set F is denoted by $\varphi * F$, and is defined as the clause-set obtained from F by first removing all clauses satisfied by φ , and then removing from the remaining clauses all literal occurrences which are falsified by φ . Partial assignments φ, ψ can be combined by the operation $\varphi \circ \psi$ which is the partial assignment with domain the union of the domains of φ and ψ , while $(\varphi \circ \psi)(v)$ is defined as $\psi(v)$ if possible and otherwise $\varphi(v)$. Using $\langle \rangle$ for the empty partial assignment, we have the following fundamental laws:

$$\begin{aligned} (\varphi \circ \psi) \circ \theta &= \varphi \circ (\psi \circ \theta) \\ \varphi \circ \langle \rangle &= \langle \rangle \circ \varphi = \varphi \\ \varphi * (\psi * F) &= (\varphi \circ \psi) * F \\ \langle \rangle * F &= F \\ \varphi * \top &= \top \\ \varphi * (F_1 \cup F_2) &= \varphi * F_1 \cup \varphi * F_2 \\ \perp \in F &\Rightarrow \perp \in \varphi * F. \end{aligned}$$

A clause-set F is satisfiable if it has a satisfying assignment φ , that is, fulfilling $\varphi * F = \top$ (while a “falsifying assignment” fulfils $\perp \in \varphi * F$). We need some specific notions to fix the variables involved in satisfying assignments:

Definition 14.8.1. For a clause-set F and a set of variables V with $\text{var}(F) \subseteq V$ let $\text{mod}_V(F)$ be the set of satisfying assignments φ with domain exactly V , while we use $\text{mod}(F) := \text{mod}_{\text{var}(F)}(F)$.

Satisfying partial assignments in the sense of Definition 14.8.1, with a fixed domain including all variables in F , are called “total satisfying assignments”. We

²Often partial assignments are identified with clauses, interpreting them as setting their literals to 1 (for CNF); for a systematic treatment however this convenient “trick” is better avoided, and we put emphasis on the different roles played by clauses and partial assignments.

denote the restriction of a partial assignment φ to the domain $V \cap \text{var}(\varphi)$ by $\varphi|V$ for a set V of variables. Finally, for a partial assignment φ we denote by C_φ^ε for $\varepsilon \in \{0, 1\}$ the clause consisting of exactly the literals x with $\varphi(x) = \varepsilon$; thus C_φ^0 consists of the literals falsified by φ , the “CNF representation” of φ , while C_φ^1 consists of the literals satisfied by φ , the “DNF representation” of φ .

Partial assignments assign truth values 0, 1 to values, and their operation on clause-sets F by $\varphi * F$ handles literals accordingly; a simpler operation is that by a (finite) set V of variables, where now literals involving these variables are simply crossed out, and we write $V * F$ for this operation. While for partial assignments the above laws just reflect, that we have an operation of the monoid of partial assignments (see Subsection 14.9.1) on the (upper) semilattice of clause-sets, here now we have the operation of finite sets of variables as (upper) semilattice, with set-union as composition, on the (upper) semilattice of clause-sets, with the laws $V * (W * F) = (V \cup W) * F$, $\emptyset * F = F$, $V * \top = \top$, $V * (F_1 \cup F_2) = V * F_1 \cup V * F_2$, and also $\perp \in F \Rightarrow \perp \in V * F$.

In order to give some examples, we need some notation for specific partial assignments, and we use for example “ $\langle x \rightarrow 1, y \rightarrow 0 \rangle$ ” to denote the partial assignment which sets variable x to 1 (i.e., true) and y to 0 (i.e., false); instead of variables also literals can be used here.

Example 14.8.1. Using four (different) variables a, b, x, y and the clause-set $F := \{\{a, b\}, \{x, \bar{a}\}, \{y, a, b\}, \{\bar{x}, y, b\}\}$ we have $\langle x \rightarrow 1, y \rightarrow 0 \rangle * F = \{\{a, b\}, \{b\}\}$ and $\{a, x\} * F = \{\{b\}, \perp, \{y, b\}\}$.

Often we need to select clauses containing some variables from a given set of variables, and we do this by $F_V := \{C \in F : \text{var}(C) \cap V \neq \emptyset\}$ for clause-sets F and sets of variables V . Finally we need to *restrict* clause-sets F to some set V of variables, and, generalising a standard notion from hypergraph theory, we use $F[V]$ for this operation, defined by

$$F[V] := (\text{var}(F) \setminus V) * F_V.$$

Note that we have $F[V] = ((\text{var}(F) \setminus V) * F) \setminus \{\perp\}$. Finally we remark that for more combinatorially-oriented investigations often clauses need to occur several times, and thus (at least) *multi*-clause-sets are needed; the reader finds the generalisation of the considerations here to multi-clause-sets in [Kul07a], while in Subsection 14.11.3.2 we discuss the more general notion of “labelled clause-sets”.

14.8.2. Autarkies

Now we come to the fundamental notion of an “autarky”. The term was coined by [MS85] for a partial assignment whose application produces a sub-clause-set, while implicitly used in works like the similar [Luc84] or in the earlier [EIS76]. Since these works took a more “logical” point of view (they were just concerned with deciding satisfiability, not with more detailed structural investigations), the underlying notion of an “autarky” was ambiguous w.r.t. whether contractions of clauses after application of partial assignments was taken into account or not. This ambiguity is resolved by distinguishing between “weak autarkies” and “(normal) autarkies”:

Definition 14.8.2. A partial assignment φ is called a **weak autarky for F** if $\varphi * F \subseteq F$ holds, while φ is an **autarky for F** if φ is a weak autarky for all sub-clause-sets F' of F .

Every satisfying assignment for F is an autarky for F . Thus every partial assignment is an autarky for \top .

Example 14.8.2. The partial assignment $\langle b \rightarrow 0 \rangle$ is a weak autarky for $F := \{\{a\}, \{a, b\}\}$, but not an autarky for F . The partial assignment $\langle b \rightarrow 1 \rangle$ is an autarky for F .

The basic fact about (weak) autarkies is the observation:

*If φ is a weak autarky for F , then $\varphi * F$ is satisfiability equivalent to F .*

(For every partial assignment φ we have that from $\varphi * F$ being satisfiable follows F being satisfiable, while the other direction, that if F is satisfiable then $\varphi * F$ is satisfiable, follows from $\varphi * F \subseteq F$.) A well-known reduction for SAT-solving is the elimination of pure literals, and this is a special case of an “autarky reduction” (see Subsection 14.8.3 for the treatment of full autarky reduction).

Example 14.8.3. A *pure literal* (also called “monotone literal”) for a clause-set F is a literal x such that \bar{x} does not occur in F (i.e., $\bar{x} \notin \bigcup F$). Every pure literal x for F yields an autarky $\langle x \rightarrow 1 \rangle$ for F . In Example 14.8.2, literals a, b are pure literals for F . See Subsection 14.11.1 for more on “pure autarkies”.

Autarkies are easily characterised by

φ is an autarky for F if and only if for every clause $C \in F$
 either φ does not “touch” C , i.e., $\text{var}(\varphi) \cap \text{var}(C) = \emptyset$,
 or φ satisfies C , i.e., $\varphi * \{C\} = \top$.

Weak autarkies are allowed to touch clauses without satisfying them, if the resulting clause (after removal of falsified literals) was already present in the original clause-set. Since we regard it as essential that an autarky for a clause-set is also an autarky for every sub-clause-set, the fundamental notion is that of an “autarky”; see Subsection 14.8.4 for a discussion. The basic characterisations of autarkies in terms of satisfying assignments are given by the following equivalent statements (for arbitrary clause-sets F and partial assignments φ):

1. φ is an autarky for F
2. iff φ is a satisfying assignment for $F_{\text{var}(\varphi)}$
3. iff φ is a satisfying assignment for $F[\text{var}(\varphi)]$.

The following properties of autarkies are considered to be the most basic ones (leading to an “axiomatic” generalisation in Section 14.11), for arbitrary clause-sets F , partial assignments φ and sets V of variables:

1. $\langle \rangle$ is an autarky for F . In general, φ is an autarky for F iff the restriction $\varphi|_{\text{var}(F)}$ is an autarky for F , and φ is called a *trivial autarky* for F if $\text{var}(\varphi) \cap \text{var}(F) = \emptyset$ (i.e., iff $\varphi|_{\text{var}(F)} = \langle \rangle$).
2. If φ is an autarky for F , then φ is also an autarky for $F' \subseteq F$.

3. If φ, ψ are autarkies for F then also $\psi \circ \varphi$ is an autarky for F . More generally, if φ is an autarky for F and ψ is an autarky for $\varphi * F$, then $\psi \circ \varphi$ is an autarky for F .
4. If $\text{var}(\varphi) \cap V = \emptyset$, then φ is an autarky for $V * F$ iff φ is an autarky for F .
5. φ is an autarky for F iff φ is an autarky for $F \cup \{\perp\}$.
6. If φ is an autarky for F , and F' is isomorphic to F (by renaming variables and flipping polarities; compare Subsection 14.9.5), then applying the same renamings and flipping of polarities to φ we obtain an autarky φ' for F' .

An *autark subset* (or “autark sub-clause-set”) of F is some $F' \subseteq F$ which is (exactly) satisfied by some autarky φ for F , i.e., $F' = F \setminus (\varphi * F)$. \top is an autarky sub-clause-set of F , and if F_1, F_2 are autark sub-clause-sets of F , then so is $F_1 \cup F_2$. It follows that there is the *largest autark sub-clause-set* of F .

14.8.3. Autarky reduction and lean clause-sets

Given an autarky φ for clause-set F , we can reduce F satisfiability-equivalent to $\varphi * F$. A clause-set F is called *lean* if no autarky-reduction is possible, that is, if every autarky for F is trivial. This notion was introduced in [Kul00b], and further studied in [Kul03, Kul07a]. Examples for lean clause-sets are \top , all minimally unsatisfiable clause-sets, and clause-sets obtained from lean clause-sets by the extension rule of extended resolution (see Lemma 3.2 in [Kul00b]). Except of \top , every lean clause-set is unsatisfiable, and F is lean iff $F \cup \{\perp\}$ is lean. If F is lean, then so is $V * F$ and the restriction $F[V]$. The union of lean clause-sets is again lean, and thus, as with autark subsets, the lean sub-clause-sets of a clause-set form a set stable under union, with smallest element \top , while the largest element is called the **lean kernel** and denoted by $N_a(F)$.

Another possibility to define the lean kernel is by using that autarky reduction is confluent: By repeating autarky-reduction for a clause-set F as long as possible, i.e., until we arrive at a lean sub-clause-set $F' \subseteq F$, we obtain the lean kernel $N_a(F) = F'$ (here the letter “N” stands for “normal form”). The lean kernel can be characterised in many other ways:

1. $N_a(F)$ is the largest lean sub-clause-set of F .
2. $F \setminus N_a(F)$ is the largest autark subset of F .
3. The decomposition $F = N_a(F) \cup (F \setminus N_a(F))$ is characterised as the unique decomposition $F = F_1 \cup F_2$, $F_1 \cap F_2 = \emptyset$, such that
 - (a) F_1 is lean, and
 - (b) $\text{var}(F_1) * F_2$ is satisfiable.
4. So $N_a(F)$ is the unique $F' \subseteq F$ such that F' is lean and $\text{var}(F') * (F \setminus F')$ is satisfiable.

The operator N_a is a kernel operator, that is, we have $N_a(F) \subseteq F$, $F \subseteq G \Rightarrow N_a(F) \subseteq N_a(G)$, and $N_a(N_a(F)) = N_a(F)$; furthermore we have $N_a(F) = \top$ iff F is satisfiable, while $N_a(F) = F$ iff F is lean.

We now consider the fundamental *duality* between autarkies and resolution. A precursor of this duality in the context of tableaux calculi has been established in [Van99] (see especially Theorems 5.2, 5.3 there); in Subsection 14.10.2 we will

further discuss this work. We recall the resolution rule, which allows for “parent clauses” C, D which clash in exactly one literal x , i.e., $C \cap \overline{D} = \{x\}$, to derive the “resolvent” $R := (C \setminus \{x\}) \cup (D \setminus \{\overline{x}\})$. A resolution refutation for F is a (rooted) binary tree labelled with clauses, such that the leaves are labelled with clauses from F , every inner node is labelled by a resolvent of its two children, and the root is labelled with \perp . F is unsatisfiable iff it has a resolution refutation. We remark that such resolution refutations do not allow for “dead ends”, and that handling “full resolution”, i.e., resolution in dag-form, where clauses can be parent-clauses for several resolution steps, can be achieved in this framework by not counting nodes of the tree but only counting different clauses. A basic observation is that a clause $C \in F$ containing a pure literal cannot be used (i.e., label a leaf) in any resolution refutation, since we can never get rid off this pure literal. This can be generalised to the fact that if φ is an autarky for F satisfying some leaf clause of a resolution tree, then φ also satisfies every clause on the path from this leaf to the root (the autarky condition prevents the possibility that all satisfied literals vanish together). Thus a clause satisfied by some autarky cannot be part of any resolution refutation. In Section 3.3 in [Kul00b] it was shown that also the reverse direction holds.

Theorem 14.8.1. *For every clause-set F the lean kernel $N_a(F)$ consists of exactly the clauses from F which can be used in some (tree) resolution refutation of F (without dead-ends). In other words, there exists an autarky satisfying some clause $C \in F$ if and only if C cannot be used in any resolution refutation of F .*

Theorem 14.8.1 yields a method for computing the lean kernel, which will be discussed in Subsection 14.10.3.

14.8.4. Remarks on autarkies vs. weak autarkies

A weak autarky φ for a clause-set F is not an autarky for F iff there exists some $F' \subset F$ such that φ is not a weak autarky for F' ; e.g., in Example 14.8.2 $\langle b \rightarrow 0 \rangle$ is not a weak autarky for $\{\{a, b\}\} \subset F = \{\{a\}, \{a, b\}\}$. So the property of being a weak autarky is not inherited by sub-clause-sets, which makes weak autarkies a more fragile, “accidental” or “non-algebraic” concept. Nevertheless, weak autarkies share many properties with autarkies:

1. If φ is a weak autarky for F , and ψ a weak autarky for $\varphi * F$, then $\psi \circ \varphi$ is a weak autarky for F (this follows by definition).
2. Consider a weak autarky ψ for F ; as mentioned, if ψ is not an autarky for F then ψ is not a weak autarky for some sub-clause-set F' of F . However ψ is a weak autarky for each $F' \subseteq F$ such that there is a weak autarky φ for F with $F' = \varphi * F$ (since if ψ is not a weak autarky for F' anymore, then the “absorbing” clause $C \in F$ for some “new” clause $\psi * D = C$, $D \in F'$, created by ψ , must have vanished, which in this case means that $\varphi * \{C\} = \top$, implying $\varphi * \{D\} = \top$, and thus also D vanished).
3. It follows that the composition of weak autarkies is again a weak autarky. So the weak autarkies for a clause-set F form a sub-monoid of $(\text{PASS}, \circ, \langle \rangle)$, containing the autarky monoid $\text{Auk}(F)$ (see Section 14.9 for the autarky monoid).

4. Furthermore reduction by weak autarkies is confluent, yielding a (well-defined) sub-clause-set of $N_a(F)$.

14.9. The autarky monoid

We have already mentioned that the composition of autarkies yields again an autarky. This fundamental observation was made in [Oku00], and it initiated the study of the *autarky monoid* in [Kul00b], continued in [KMT08]. In this subsection we discuss the basic features of the autarky monoid. Some prerequisites from algebra are used, and the following literature might help to provide the missing (always elementary) definitions: [Lau06] in Chapters 1 - 4 provides basic material on algebraic structures and hull (“closure”) systems; [Bou89], Sections §I.1 - §I.5, presents the basics on monoids, groups and operations in a systematic way, while Chapter I in [Lan02] presents such material (including categories) in a somewhat more advanced fashion; regarding category theory an accessible introduction is given by the first two chapters of [Pie91].

14.9.1. The monoid of partial assignments

Since “autarky monoids” are sets of autarkies together with the composition of partial assignments, we need to take a closer look at the monoid \mathcal{PASS} of all partial assignments. Starting with an arbitrary set \mathcal{VA} of “variables”, we obtain the set \mathcal{LIT} of literals (uncomplemented and complemented variables), the set \mathcal{CL} of clauses (finite and clash-free sets of literals) and the set \mathcal{CLS} of clause-sets (finite sets of clauses), while \mathcal{PASS} is the set of maps $\varphi : V \rightarrow \{0, 1\}$ for some finite $V \subseteq \mathcal{VA}$.³ The monoid $(\mathcal{PASS}, \circ, \langle \rangle)$ of partial assignments (together with the composition of partial assignments, and the empty partial assignment as the neutral element) has the following properties:

1. \mathcal{PASS} is generated by the elementary partial assignments $\langle v \rightarrow \varepsilon \rangle$ for variables $v \in \mathcal{VA}$ and $\varepsilon \in \{0, 1\}$. The defining relations between elementary partial assignments (those relations which specify the composition of \mathcal{PASS}) are

- (a) $\langle v \rightarrow \varepsilon \rangle \circ \langle v \rightarrow \varepsilon' \rangle = \langle v \rightarrow \varepsilon' \rangle$;
- (b) $\langle v \rightarrow \varepsilon \rangle \circ \langle w \rightarrow \varepsilon' \rangle = \langle w \rightarrow \varepsilon' \rangle \circ \langle v \rightarrow \varepsilon \rangle$ for $v \neq w$.

For a partial assignment $\varphi \in \mathcal{PASS}$ we have the unique representation $\varphi = \circ_{v \in \text{var}(\varphi)} \langle v \rightarrow \varphi(v) \rangle$ (using the commutativity of elementary partial assignments for different variables).

³The set \mathcal{VA} may be some finite set; more useful would be the choice $\mathcal{VA} = \mathbb{N}$, while actually for a theoretical study the choice of \mathcal{VA} as a universe of set theory (a set stable under all set-theoretical operations) is most appropriate. Literals are best represented via $\mathcal{LIT} := \mathcal{VA} \times \{0, 1\}$, where 0 stands for “uncomplemented”. And a map f with domain X is a set $\{(x, f(x)) : x \in X\}$ of ordered pairs, so that a partial assignment is formally the same as a clause, namely a set of pairs (v, ε) , only that now the interpretation is the opposite: while a literal $(v, \varepsilon) \in C$ in a clause C means “ v shall not get value ε ”, an assignment $(v, \varepsilon) \in \varphi$ for a partial assignment φ means that v gets value ε ; this duality reflects that clauses are part of a CNF representation of the underlying boolean function, while (satisfying) partial assignments are part of a DNF representation.

2. The closure (hull) of a finite set $P \subseteq \mathcal{PASS}$ of partial assignments under composition (i.e., the generated monoid) is again finite (since every partial assignment involves only finitely many variables).
3. \mathcal{PASS} is idempotent, i.e., for all $\varphi \in \mathcal{PASS}$ we have $\varphi \circ \varphi = \varphi$.
4. The following assertions are equivalent for $\varphi, \psi \in \mathcal{PASS}$:

- (a) φ, ψ commute (i.e., $\varphi \circ \psi = \psi \circ \varphi$);
- (b) φ, ψ are compatible (i.e., $\forall v \in \text{var}(\varphi) \cap \text{var}(\psi) : \varphi(v) = \psi(v)$);
- (c) there exists a partial assignment $\theta \in \mathcal{PASS}$ with $\varphi \subseteq \theta$ and $\psi \subseteq \theta$.

For a partial assignment $\varphi \in \mathcal{PASS}$ the submonoid $\mathcal{PASS}(\varphi) := \{\psi \in \mathcal{PASS} : \psi \subseteq \varphi\}$ of partial assignments contained in φ is commutative, and for every finite commutative submonoid M of \mathcal{PASS} there exists a partial assignment φ with $M \subseteq \mathcal{PASS}(\varphi)$.

5. The partial order \subseteq between partial assignments⁴ has the following properties:

- (a) $\langle \rangle$ is the smallest element; there is no largest element iff \mathcal{VA} is not empty, and there are no maximal elements iff \mathcal{VA} is infinite.
- (b) The infimum of any set $\emptyset \neq P \subseteq \mathcal{PASS}$ of partial assignments is the intersection of the elements of P .
- (c) $P \subseteq \mathcal{PASS}$ has an upper bound iff P is finite and all elements of P are pairwise compatible, and in this case P has a supremum (namely the union).
- (d) The order is right-compatible with the composition, that is, for partial assignments $\varphi, \psi, \theta \in \mathcal{PASS}$ with $\varphi \subseteq \psi$ we have $\varphi \circ \theta \subseteq \psi \circ \theta$.

However, the order is not left-compatible with the composition, so for example we have $\langle v \rightarrow 0 \rangle \supset \langle \rangle$, but $\langle v \rightarrow 1 \rangle \circ \langle v \rightarrow 0 \rangle = \langle v \rightarrow 0 \rangle \not\subseteq \langle v \rightarrow 1 \rangle \circ \langle \rangle = \langle v \rightarrow 1 \rangle$.

- (e) The order is identical with the natural right-compatible order of the monoid \mathcal{PASS} , that is, $\varphi \subseteq \psi \Leftrightarrow \psi \circ \varphi = \psi$ (while $\text{var}(\varphi) \subseteq \text{var}(\psi) \Leftrightarrow \varphi \circ \psi = \psi$).
- (f) Since the smallest element is the neutral element, \mathcal{PASS} is “right non-negative”, and we always have $\varphi \subseteq \psi \circ \varphi$ (while we only have $\text{var}(\varphi) \subseteq \text{var}(\varphi \circ \psi)$).

6. The formation of the domain of partial assignments, i.e., the map var , is a (surjective) homomorphism from the monoid \mathcal{PASS} to the monoid of finite subsets of \mathcal{VA} together with union, that is, $\text{var}(\varphi \circ \psi) = \text{var}(\varphi) \cup \text{var}(\psi)$ and $\text{var}(\langle \rangle) = \emptyset$. Restricted to the commutative submonoid $\mathcal{PASS}(\varphi)$ for some $\varphi \in \mathcal{PASS}$, var is an isomorphism from $\mathcal{PASS}(\varphi)$ to $\mathbb{P}(\text{var}(\varphi))$ (the powerset of the domain of φ).

A natural representation of \mathcal{PASS} as a transformation monoid is obtained as follows: Assume $0, 1 \notin \mathcal{VA}$, let $\mathcal{VA}' := \mathcal{VA} \cup \{0, 1\}$, and define the operation $*$: $\mathcal{PASS} \times \mathcal{VA}' \rightarrow \mathcal{VA}'$ by $\varphi * v := \varphi(v)$ for $v \in \text{var}(\varphi)$, and $\varphi * x := x$ otherwise. Now we have $\langle \rangle * x = x$ and $(\varphi \circ \psi) * x = \varphi * (\psi * x)$ for $x \in \mathcal{VA}'$.

⁴ $\varphi \subseteq \psi$ holds iff $\text{var}(\varphi) \subseteq \text{var}(\psi)$ and $\forall v \in \text{var}(\varphi) : \varphi(v) = \psi(v)$; we have $\varphi \subseteq \psi$ iff $C_\varphi^0 \subseteq C_\psi^0$ iff $C_\varphi^1 \subseteq C_\psi^1$

Furthermore the operation is faithful, i.e., for $\varphi \neq \psi$ there is $x \in \mathcal{VA}'$ with $\varphi * x \neq \psi * x$. The corresponding injective monoid-morphism from \mathcal{PASS} into the set of transformations of \mathcal{VA}' represents partial assignments φ and their composition by transformations $(\varphi * x)_{x \in \mathcal{VA}'} : \mathcal{VA}' \rightarrow \mathcal{VA}'$ and their composition.

14.9.2. Autarky monoid and autarky semigroup

As already mentioned, the composition of autarkies is again an autarky, which can be easily seen as follows: Consider autarkies φ, ψ for F , that is, for all $F' \subseteq F$ we have $\varphi * F' \subseteq F'$ and $\psi * F' \subseteq F'$. Now we have $(\varphi \circ \psi) * F' = \varphi * (\psi * F') \subseteq \varphi * F'' \subseteq F''$ for $F'' := \psi * F' \subseteq F'$, and thus $(\varphi \circ \psi)$ is an autarky for F .

Definition 14.9.1. For a clause-set $F \in \mathcal{CLS}$ the **autarky monoid**, denoted by $\text{Auk}(F)$, is the sub-monoid of \mathcal{PASS} given by all autarkies for F , while the **autarky semigroup** is $\text{Auks}(F) := \text{Auk}(F) \setminus \{\langle \rangle\}$. For performing computations, the **restricted** versions $\text{Auk}^r(F) := \{\varphi \in \text{Auk}(F) : \text{var}(\varphi) \subseteq \text{var}(F)\}$ and $\text{Auks}^r(F) := \text{Auks}(F) \setminus \{\langle \rangle\}$ are preferable.

Remarks:

1. $\text{Auk}(F), \text{Auk}^r(F)$ are submonoids of \mathcal{PASS} , where $\text{Auk}^r(F)$ is always finite, while $\text{Auk}(F)$ is infinite iff \mathcal{VA} is infinite. $\text{Auks}(F), \text{Auks}^r(F)$ are subsemigroups of $\mathcal{PASS} \setminus \{\langle \rangle\}$.
2. $\text{Auk}(F)$ is obtained from the elements of $\text{Auk}^r(F)$ by extending them arbitrarily on variables not in $\text{var}(F)$.
3. For every set X the “right-null semigroup” is the semigroup $(X, *)$ with $x * y := y$. Now the set $\text{mod}(F)$ of total satisfying assignments for a clause-set F is a right-null sub-semigroup of $\text{Auks}^r(F)$, and if F is satisfiable, then this set equals the set of “maximal autarkies”, as discussed below.

Some simple examples, where we know the full autarky monoid:

1. F is lean iff $\text{Auk}^r(F) = \{\langle \rangle\}$ iff $\text{Auks}^r(F) = \emptyset$.
2. Consider the unique clause-set F with $\text{var}(F) = \{v_1, \dots, v_n\}$ ($n \geq 1$) which is equivalent (as CNF) to the condition $v_1 \oplus \dots \oplus v_n = \varepsilon \in \{0, 1\}$, where “ \oplus ” is exclusive-or (F consists of 2^{n-1} clauses, each involving all variables). Now every non-trivial autarky for F is a satisfying assignment, and every satisfying assignment involves all variables, and so the restricted autarky semigroup of F is the right-null semigroup on the set of the 2^{n-1} satisfying assignments.

Basic properties of the autarky monoid are as follows:

1. Let $\text{var}(\text{Auk}(F)) := \text{var}(\text{Auk}^r(F)) := \bigcup_{\varphi \in \text{Auk}^r(F)} \text{var}(\varphi)$. Then the largest autark subset of F is $F_{\text{var}(\text{Auk}(F))}$.
2. Since $\text{Auk}^r(F)$ is finite, $\text{Auk}^r(F)$ has maximal elements, called *maximal autarkies*, and for every maximal autarky φ we have $\text{var}(\varphi) = \text{var}(\text{Auk}(F))$, while $F \setminus (\varphi * F)$ is the largest autark subset of F . We denote the set of maximal autarkies of F by $\text{Auk}\uparrow(F)$. Note that F is lean iff $\text{Auk}\uparrow(F) = \{\langle \rangle\}$.
3. The maximal autarkies of F are exactly the total satisfying assignments for $F[\text{var}(\text{Auk}(F))]$. So $\text{Auk}\uparrow(F)$ is a right-null sub-semigroup of $\text{Auks}^r(F)$. If F is satisfiable, then $\text{Auk}\uparrow(F) = \text{mod}(F)$.

4. The minimal elements of the restricted autarky semigroup $\text{Auks}^r(F)$ are called *minimal autarkies*. F has a minimal autarky iff F is not lean, and the minimal autarkies are exactly the atoms of the partial order $\text{Auk}^r(F)$. (One could thus speak of “atomic autarkies”, but it seems that the slightly different contexts for “maximal” and “minimal” autarkies do not cause confusion.) We denote the set of minimal autarkies of F by $\text{Auk}\downarrow(F)$ (so F is lean iff $\text{Auk}\downarrow(F) = \emptyset$).

If only the autark *subsets* are of interest (not their certificates, i.e., their associated autarkies), then the (upper) semilattice (with zero)

$$\text{Auk}^s(F) := \{F' \subseteq F \mid \exists \varphi \in \text{Auk}(F) : \varphi * F = F \setminus F'\}$$

of autark subsets is to be studied (the operation is just set-union, the neutral element is \top). The canonical homomorphism $\Phi : \text{Auk}(F) \rightarrow \text{Auk}^s(F)$ given by $\varphi \mapsto F \setminus (\varphi * F)$ is surjective, and the inverse image of \top is the set of trivial autarkies. Also the restriction $\Phi : \text{Auk}^r(F) \rightarrow \text{Auk}^s(F)$ is surjective, and in general also the restriction is not injective.

14.9.3. Finding (semi-)maximal autarkies

Given any nontrivial autarky, we can efficiently find some (contained) minimal autarky by the observation made in [KMT08], that “within” a partial assignment autarky search is easy:

Definition 14.9.2. For any submonoid M resp. subsemigroup S of \mathcal{PASS} and for a partial assignment $\varphi \in \mathcal{PASS}$ we denote by $M(\varphi) := M \cap \mathcal{PASS}(\varphi)$ resp. $S(\varphi) := S \cap \mathcal{PASS}(\varphi)$ the submonoid resp. subsemigroup of M resp. S obtained by restriction to partial assignments contained in φ .

Lemma 14.9.1. Consider a clause-set $F \in \mathcal{CLS}$ and a partial assignment $\varphi \in \mathcal{PASS}$. Then $\text{Auk}(F)(\varphi)$ has a (unique) largest element φ_0 , which can be efficiently computed as follows:

1. Let $\varphi_0 := \varphi$.
2. If φ_0 is an autarky for F , then stop. Otherwise, choose $C \in F$ with $\text{var}(\varphi) \cap \text{var}(C) \neq \emptyset$ and $\varphi * \{C\} \neq \top$, choose $v \in \text{var}(\varphi) \cap \text{var}(C)$, let φ_0 be the restriction of φ_0 to $\text{var}(\varphi_0) \setminus \{v\}$, and repeat this step.

For a given clause-set F with $V := \text{var}(F)$, the map $\varphi \in \{\varphi \in \mathcal{PASS} : \text{var}(\varphi) \subseteq V\} \mapsto \varphi_0 \in \text{Auk}^r(F)$, which assigns to every partial assignment the contained largest autarky, is obviously surjective. Such autarkies φ_0 which are largest elements of some $\text{Auk}(F)(\varphi)$ for total assignments φ (i.e., $\text{var}(\varphi) = \text{var}(F)$) are called *semi-maximal autarkies*. In Subsection 14.10.1 some applications are discussed.

Corollary 14.9.2. Given a nontrivial autarky φ for a clause-set F , we can efficiently compute some minimal autarky $\varphi_0 \in \text{Auk}\downarrow(F)$ (with $\varphi_0 \subseteq \varphi$).

Proof. Try removing assignments from φ and obtaining a smaller nontrivial autarky by Lemma 14.9.1 until a minimal autarky is obtained. \square

14.9.4. Generating autarkies

Now we are considering generating sets of the autarky monoid, i.e., subsets E of $\text{Auk}^r(F)$ such that the generated submonoid is $\text{Auk}^r(F)$ itself. The autarky monoid has the potential to give a succinct representation of a large set of satisfying assignments: Though there are even more autarkies than satisfying assignments, a generating set might nevertheless be much smaller.

Example 14.9.1. For $n \geq 0$ we consider the (well-known) satisfiable clause-set $F := \{\{v_1, w_1\}, \dots, \{v_n, w_n\}\}$ with $2n$ variables (well-known for not having a short DNF-representation). The structure of $\text{Auk}^r(F)$ is as follows:

- There are $2n$ minimal autarkies $\langle v_i \rightarrow 1 \rangle, \langle w_i \rightarrow 1 \rangle$.
- $|\text{mod}(F)| = 3^n$, where the satisfying (total) assignments, which are also the maximal autarkies here, are all combinations of the three satisfying assignments for the clauses $\{v_i, w_i\}$.
- $|\text{Auk}^r(F)| = 6^n$, where the autarkies are all combinations of the six autarkies for the clauses $\{v_i, w_i\}$.
- Using $E_i := \{\langle v_i \rightarrow 1 \rangle, \langle w_i \rightarrow 1 \rangle, \langle v_i \rightarrow 1, w_i \rightarrow 0 \rangle, \langle w_i \rightarrow 1, v_i \rightarrow 0 \rangle\}$, the set $E := \bigcup_{i=1}^n E_i$ is a generating set for $\text{Auk}^r(F)$ of size $4n$.

So a basic question is to obtain interesting notions of generating sets for autarky monoids. Only at first glance it might appear that the minimal autarkies are sufficient to compute at least the lean kernel, but the following example shows that this is clearly not the case.

Example 14.9.2. Minimally unsatisfiable Horn clause-sets have deficiency 1; moreover, the Horn clause-sets with $n \geq 0$ variables, which are even saturated minimally unsatisfiable, are up to the names of the variables exactly the clause-sets $H_n := H'_n \cup \{\{\bar{v}_1, \dots, \bar{v}_n\}\}$, where

$$H'_n := \{\{v_1\}, \{\bar{v}_1, v_2\}, \{\bar{v}_1, \bar{v}_2, v_3\}, \dots, \{\bar{v}_1, \dots, \bar{v}_{n-1}, v_n\}\}.$$

For example $H_0 = \{\perp\}$, $H_1 = \{\{v_1\}, \{\bar{v}_1\}\}$ and $H_2 = \{\{v_1\}, \{\bar{v}_1, v_2\}, \{\bar{v}_1, \bar{v}_2\}\}$. It is not hard to see that

$$\text{Auk}^r(H'_n) = \{\langle \rangle, \langle v_n \rightarrow 1 \rangle, \langle v_{n-1}, v_n \rightarrow 1 \rangle, \dots, \langle v_1, \dots, v_n \rightarrow 1 \rangle\}.$$

So $\text{Auk}^r(H'_n)$ as a partial order is a linear chain with $n + 1$ elements, with $\langle \rangle$ as smallest element and with $\varphi_n := \langle v_1, \dots, v_n \rightarrow 1 \rangle$ as largest element; for example $\text{Auk}^r(H'_2) = \{\langle \rangle, \langle v_2 \rightarrow 1 \rangle, \langle v_1, v_2 \rightarrow 1 \rangle\}$. For $n \geq 1$ the only minimal autarky is $\langle v_n \rightarrow 1 \rangle$.

So what can we say about a generating set G of the (restricted) autarky semigroup $\text{Auks}^r(F)$? Clearly every minimal autarky for F must be element of G , and, more generally, every *directly indecomposable* autarky $\varphi \in \text{Auks}^r(F)$ must be in G , which are characterised by the condition that whenever $\varphi = \psi_1 \circ \psi_2$ for $\psi_1, \psi_2 \in \text{Auk}^r(F)$ holds, then $\psi_1 = \varphi$ or $\psi_2 = \varphi$ must be the case. The directly indecomposable autarkies for Example 14.9.1 are exactly the elements of the generating set E given there, but the following example shows that the directly indecomposable autarkies in general do not generate all autarkies.

Example 14.9.3. Let $F := \{\{v_1, v_2, v_3\}, \{v_1, \bar{v}_2, \bar{v}_3\}\}$. For a partial assignment φ and a set of variables V with $V \cap \text{var}(\varphi) = \emptyset$ let us denote by $[\varphi]_V$ the set of all partial assignments ψ with $\varphi \subseteq \psi$ and $\text{var}(\psi) \subseteq \text{var}(\varphi) \cup V$. Using $\varphi_1 := \langle v_1 \rightarrow 1 \rangle$, $\varphi_2 := \langle v_2 \rightarrow 1, v_3 \rightarrow 0 \rangle$ and $\varphi_3 := \langle v_2 \rightarrow 0, v_3 \rightarrow 1 \rangle$ we have $\text{Auks}^r(F) = [\varphi_1]_{\{v_2, v_3\}} \cup [\varphi_2]_{\{v_3\}} \cup [\varphi_3]_{\{v_3\}}$, whence $|\text{Auks}^r(F)| = 3^2 + 3 + 3 - 1 - 1 = 13$. There are three minimal autarkies, namely $\varphi_1, \varphi_2, \varphi_3$, and four further directly indecomposable autarkies, namely $\langle v_1 \rightarrow 1, v_2 \rightarrow \varepsilon \rangle$ and $\langle v_1 \rightarrow 1, v_3 \rightarrow \varepsilon \rangle$ for $\varepsilon \in \{0, 1\}$. These seven elements generate eleven of the thirteen (non-trivial) autarkies, leaving out $\varphi_2 \circ \langle v_1 \rightarrow 0 \rangle$ and $\varphi_3 \circ \langle v_1 \rightarrow 0 \rangle$.

Given a set $E \subseteq \text{Auk}^r(F)$ of autarkies, instead of allowing all compositions of elements of E it makes sense to only allow “commutative compositions”, that is we only consider subsets $E' \subseteq E$ of pairwise commuting (pairwise compatible) partial assignments and their composition $\circ_{\varphi \in E'} \varphi$; note that due to the imposed commutativity here we do not need to take care of the order, and due to idempotency also no exponents need to be considered. We call E a *commutatively generating set* for $\text{Auks}^r(F)$ if the set of commutative compositions of E is $\text{Auks}^r(F)$, while we speak of a *commutatively semi-generating set*, if the commutative compositions of E at least yield all semi-maximal autarkies (different from $\langle \rangle$; recall Subsection 14.9.3).⁵ The generating set E in Example 14.9.1 is also a commutatively generating set; Example 14.9.1 is just a combination of variable-disjoint clauses, and we conclude this subsection by discussing the autarky-monoid and its generation for singleton clause-sets (the smallest non-trivial examples).

Example 14.9.4. Consider a clause C , and let $F := \{C\}$ and $n := n(F) = |C|$.

- There are n minimal autarkies (i.e., $|\text{Auk}\downarrow(F)| = n$).
- $|\text{Auk}^r(F)| = 3^n - (2^n - 1)$ (the non-autarkies are all partial assignments using only value 0, except of the empty partial assignment).
- For $n \geq 1$ there are $2^n - 1$ maximal autarkies (i.e., $|\text{Auk}\uparrow(F)| = 2^n - 1$), which are also the satisfying assignments, while if $n = 0$, then F is unsatisfiable and lean.
- Every semi-maximal autarky is also a maximal autarky.
- There are $n + n(n - 1)$ directly indecomposable autarkies, namely the minimal autarkies plus the partial assignments $\langle x \rightarrow 1, y \rightarrow 0 \rangle$ for $x \in C$ and $y \in C \setminus \{x\}$.
- The set I of directly indecomposable autarkies is a (smallest) generating set, and it is also a commutatively generating set.
- If we replace the n minimal autarkies in I by the single autarky $\langle x \rightarrow 1 : x \in C \rangle$, obtaining I' , then I' is a commutatively semi-generating set (of size $n(n - 1) + 1$).

14.9.5. The autarky monoid as a functor

In [Sze03] the basic form of categories of clause-sets has been introduced (recall Section 14.3), called \mathcal{CLS} here, and which is defined as follows:

⁵We could also speak of “compatibly generating”. Using “sign vectors” instead of partial assignments, according to Definition 5.31 in [BK92] we could also speak of “conformally generating”.

1. The set of objects of \mathfrak{CLS} is \mathcal{CLS} .
2. For a clause-set F let $\text{lit}(F) := \text{var}(F) \cup \overline{\text{var}(F)}$ be the set of literals over F (which occur in at least one polarity; for a set L of literals we use $\overline{L} := \{\bar{x} : x \in L\}$).
3. Now a morphism $f : F \rightarrow G$ from clause-set F to clause-set G is a map $f : \text{lit}(F) \rightarrow \text{lit}(G)$ with the following two properties:
 - (a) f preserves complements, i.e., for $x \in \text{lit}(F)$ we have $f(\bar{x}) = \overline{f(x)}$.
 - (b) f preserves clauses, i.e., for $C \in F$ we have $f(C) \in G$, where $f(C) := \{f(x) : x \in C\}$.

The isomorphisms in \mathfrak{CLS} are the standard isomorphisms between clause-sets, allowing the renaming of variables accompanied with flipping polarities. Extending [Sze03] (Section 3) we can now recognise $F \in \mathcal{CLS} \mapsto \text{Auk}(F)$ as a contravariant functor from \mathfrak{CLS} to the category \mathfrak{MON} of monoids (objects are monoids, morphisms are homomorphisms of monoids):

Lemma 14.9.3. *The formation of the autarky monoid is a contravariant functor $\text{Auk} : \mathfrak{CLS} \rightarrow \mathfrak{MON}$, where for a morphism $f : F \rightarrow G$ between clause-sets the homomorphism $\text{Auk}(f) : \text{Auk}(G) \rightarrow \text{Auk}(F)$ assigns to the autarky $\varphi \in \text{Auk}(G)$ the partial assignment $\text{Auk}(f)(\varphi)$ given as follows:*

1. the domain of $\text{Auk}(f)(\varphi)$ is the set of variables $v \in \text{var}(F)$ such that $\text{var}(f(v)) \in \text{var}(\varphi)$;
2. for $v \in \text{var}(\text{Auk}(f)(\varphi))$ we set $\text{Auk}(f)(\varphi)(v) := \varphi(f(v))$.

Proof. First for $\varphi \in \text{Auk}(G)$ and $\psi := \text{Auk}(f)(\varphi)$ we need to show $\psi \in \text{Auk}(F)$. So consider a clause $C \in F$ touched by ψ . Thus there is $x \in C$ such that $\varphi(f(x))$ is defined. Now φ touches $f(C)$, so there is $x' \in C$ with $\varphi(f(x')) = 1$, i.e., $\psi(x') = 1$. To see that Auk preserves identities we just note that $\text{id}_F = \text{id}_{\text{lit}(F)}$, and then $\text{Auk}(\text{id}_F) = \text{id}_{\text{Auk}(F)}$. Finally consider morphisms $f : F \rightarrow G$ and $g : G \rightarrow H$ in \mathfrak{CLS} ; we have to show $\text{Auk}(g \circ f) = \text{Auk}(f) \circ \text{Auk}(g)$, and this follows directly from the definitions. \square

Also Auk^f yields a functor. We remark that the coproduct in \mathfrak{CLS} is the variable-disjoin sum, and the functor Auk^f anti-preserved coproducts, i.e., maps the variable-disjoin sum of clause-sets to the product of their autarky-monoids.

14.9.6. Galois correspondences

Finally we consider the Galois correspondence given by the autarky notion. Consider the set \mathcal{CL} of clauses and the set \mathcal{PASS} of partial assignments. We have two fundamental relations between clauses and partial assignments:

1. S_0 is the set of pairs $(C, \varphi) \in \mathcal{CL} \times \mathcal{PASS}$ such that φ is a satisfying assignment for $\{C\}$, that is, $\varphi * \{C\} = \top$.
2. S_1 is the set of pairs $(C, \varphi) \in \mathcal{CL} \times \mathcal{PASS}$ such that φ is an autarky for $\{C\}$, that is, $\varphi * \{C\} \in \{\top, \{C\}\}$.

If we use the alternative representation $\mathcal{PASS}' := \mathcal{CL}$ of partial assignments by clauses containing the literals satisfied by the partial assignment, i.e., a partial

assignment φ corresponds to the clause C_φ^1 , then we obtain the corresponding relations $S'_0 = \{(C, D) \in \mathcal{CL} \times \mathcal{PASS}' : C \cap D \neq \emptyset\}$ and $S'_1 = \{(C, D) \in \mathcal{CL} \times \mathcal{PASS}' : C \cap \overline{D} \neq \emptyset \Rightarrow C \cap D \neq \emptyset\}$.

Given the “polarities” S_0, S_1 , we obtain (as usual) Galois correspondences (X_0, Y_0) resp. (X_1, Y_1) between the partial orders $(\mathbb{P}(\mathcal{CL}), \subseteq)$ and $(\mathbb{P}(\mathcal{PASS}), \subseteq)$, given as $X_i : \mathbb{P}(\mathcal{CL}) \rightarrow \mathbb{P}(\mathcal{PASS})$ and $Y_i : \mathbb{P}(\mathcal{PASS}) \rightarrow \mathbb{P}(\mathcal{CL})$ defined by

$$\begin{aligned} X_i(F) &:= \{\varphi \in \mathcal{PASS} \mid \forall C \in F : S_i(C, \varphi)\} \\ Y_i(P) &:= \{C \in \mathcal{CL} \mid \forall \varphi \in P : S_i(C, \varphi)\}. \end{aligned}$$

for $F \subseteq \mathcal{CL}$ and $P \subseteq \mathcal{PASS}$. Using $\text{mod}^*(F) := \{\varphi \in \mathcal{PASS} : \varphi * F = \top\}$ for the set of all satisfying partial assignments for F , we see that

$$X_0(F) = \text{mod}^*(F), \quad X_1(F) = \text{Auk}(F)$$

for sets of clauses F .⁶ As it is well known, $Z_i := Y_i \circ X_i$ for $i \in \{0, 1\}$ yields a closure operator on $\mathbb{P}(\mathcal{CL})$ (i.e., $Z_i(F) \supseteq F$, $Z_i(Z_i(F)) = Z_i(F)$ and $F \subseteq G \Rightarrow Z_i(F) \subseteq Z_i(G)$). It is a well-known generalisation of the fundamental duality-law for the formation of the transversal hypergraph, that $Z_0(F)$ is just the set of all clauses C which follow logically from F , i.e., $Z_0(F) = \{C \in \mathcal{CL} : F \models C\}$. Using \mathcal{PASS}' instead of \mathcal{PASS} , we have $Y_i = X_i$ for $i \in \{0, 1\}$, and thus the semantical closure $Z_0(F)$ can be understood as $Z_0(F) = \text{mod}^*(\text{mod}^*(F)) =: \text{mod}^{**}(F)$. And analogously we can understand the *autarky closure* $Z_1(F)$, the largest set of clauses with the same autarky monoid as F , as $Z_1(F) = \text{Auk}(\text{Auk}(F))$; thus we write the autarky closure as $\text{Auk}^2(F) := Z_1(F)$. Some simple properties of the autarky closure are (for all $F \subseteq \mathcal{CL}$):

1. $\text{Auk}^2(\top) = \top$ (also $\text{mod}^{**}(\top) = \top$).
2. $\perp \in \text{Auk}^2(F)$, and thus $\text{Auk}^2(F) \supseteq F \cup \{\perp\}$ (while $\perp \in \text{mod}^{**}(F)$ iff F is unsatisfiable).
3. We have $\text{var}(\text{Auk}^2(F)) = \text{var}(F)$. This is quite different from mod^{**} , where for $F \neq \top$ we have $\text{var}(\text{mod}^{**}(F)) = \mathcal{VA}$, since $\text{mod}^{**}(F)$ is stable under addition of super-clauses.
4. A clause-set F is lean iff $\text{Auk}^2(F) = \{C \in \mathcal{CL} : \text{var}(C) \subseteq \text{var}(F)\}$, that is, iff $\text{Auk}^2(F)$ contains all possible clauses over $\text{var}(F)$ (in other words, iff $\text{Auk}^2(F)$ is as large as possible).
5. $\text{Auk}^2(F)$ is stable under resolution, that is, if $C, D \in F$ with $C \cap \overline{D} = \{x\}$, then $R \in \text{Auk}^2(F)$ for the resolvent $R := (C \setminus \{x\}) \cup (D \setminus \{\overline{x}\})$.
6. $\text{Auk}^2(F)$ is stable under the composition of partial assignments considered as clauses, that is, if $C, D \in \text{Auk}^2(F)$ then also $(C \setminus \overline{D}) \cup D \in \text{Auk}^2(F)$.

14.10. Finding and using autarkies

In this section we discuss the methods which have been applied to find autarkies, and what use has been made of those autarkies. First some remarks on the complexity of the most basic algorithmic problems. The class of lean clause-sets

⁶This includes, of course, clause-sets, which we defined as *finite* sets of clauses.

has been shown in [Kul03] to be coNP-complete, and thus finding a non-trivial autarky is NP-complete. By Corollary 14.9.2 also finding a minimal autarky is NP-complete, while by contrast deciding whether for a given clause-set F and partial assignment φ it is φ a maximal autarky for F is coNP-complete (if φ is an autarky for F , then φ is a maximal autarky iff $F \setminus (\varphi * F)$ is lean). As shown in [KMT08], the “autarky existence problem” is self-reducible, that is, given an oracle for deciding whether a clause-set is lean, one can efficiently find a maximal autarky. Lemma 8.6 in [Kul03] shows how (just) to compute the lean kernel given such an oracle. In [KMT08] one finds complexity characterisations of other basic tasks related to autarkies. In Subsection 14.11.6 we will discuss a refined list of the basic algorithmic problems of autarky theory.

14.10.1. Direct approaches

At first sight it might appear that time $\tilde{O}(3^{n(F)})$ is needed to exhaustively search for a non-trivial autarky in a clause-set F , since there are $3^{n(F)}$ partial assignments. However, we can find all semi-maximal autarkies (recall Subsection 14.9.3) in time $\tilde{O}(2^{n(F)})$ by running through all total assignments φ and determining the maximal autarky $\varphi_0 \subseteq \varphi$ for F (this was first observed in [KMT08]). Thus the lean kernel can be computed in time $\tilde{O}(2^{n(F)})$ (for arbitrary clause-sets, using linear space). See Subsection 14.10.8 for applications of this method in local search solvers (while in Corollary 14.10.2 the bound $\tilde{O}(2^{n(F)})$ is improved for restricted clause-length).

The direct backtracking approach searching for an autarky is closely related to the tableau calculus, when focusing on a specific clause to be satisfied by some autarky, and we further comment on this subject in Subsection 14.10.2. From a somewhat more general perspective, searching for an autarky can be considered as a constraint satisfaction problems where variables have 3-valued domains; yet there are no experiences with this approach, but in Subsection 14.10.4 we discuss the reduction of the autarky search problem to SAT.

14.10.2. Autarkies and the tableau-calculus

A (standard) tableau-based SAT solver starts with a “top” clause $C \in F$, chooses a literal $x \in C$, and tries recursively to refute that $\langle x \rightarrow 1 \rangle$ could be extended to a satisfying assignment for F (this is then repeated for all literals in C , with the goal to conclude unsatisfiability of F at the end). It has been observed in [Van99] that this process should better be considered as the natural form of searching for an autarky for F satisfying C , since in case the solver gets stuck, that is, cannot finish the unsatisfiability-proof based on x (and thus cannot proceed to the other literals in C), we actually found an autarky $\langle x \rightarrow 1 \rangle \subseteq \varphi$ for F , so that we can repeat the whole process with $\varphi * F \subset F$, and either finally find F unsatisfiable (that is, find some subset $F' \subseteq F$ unsatisfiable with $N_a(F) \subseteq F'$), or can compose the various autarkies φ found in the repetitions of this process to a (global) satisfying assignment. This view of the (standard, propositional) tableau calculus yields an explanation why SAT solvers are usually faster: Exhaustively searching for an autarky is a somewhat harder task than just (exhaustively) searching for a satisfying assignment (especially to determine that there is none), and so for the

clause-based branching performed by a tableau-solver the top clause C is split into $|C|$ branches $\langle x \rightarrow 1 \rangle$ for $x \in C$, while a SAT solver can use some order $C = \{x_1, \dots, x_k\}$ and set in branch i additionally to $x_i \rightarrow 1$ also $x_j \rightarrow 0$ for $j < i$ (where for the autarky search the x_j stay unassigned). Roughly spoken, SAT only needs to search through 2^n (total) assignments, while tableau search somehow needs to search through 3^n partial assignments. However for instances having “nice” autarkies (not too small, not too big), the autarky approach should have merits. The approach from [Van99] has been extended by parallel processing in [Oku00], exploiting that the composition of autarkies again is an autarky (and so we can combine autarkies found in parallel), while applications are discussed in [VO99].

14.10.3. Computing the lean kernel by the autarky-resolution duality

Based on the duality between autarkies and resolution (recall Theorem 14.8.1), the following meta-theorem has been given (in variations) in [Kul01, Kul07a] for computing the lean kernel:

Theorem 14.10.1. *Consider a class $\mathcal{C} \subseteq \text{CLS}$ of clause-sets, an “upper bound” function $f : \mathcal{C} \rightarrow \mathbb{R}_{\geq 0}$ and an algorithm \mathfrak{A} defined on inputs $F \in \mathcal{C}$ with the following properties:*

1. \mathcal{C} is stable under restriction, that is, for all (finite) $V \subseteq \mathcal{VA}$ and all $F \in \mathcal{C}$ we have $F[V] \in \mathcal{C}$.
2. The running time of \mathfrak{A} on input $F \in \mathcal{C}$ is $\tilde{O}(f(F))$.
3. f is monotone w.r.t. restriction, that is, for $V \subseteq \mathcal{VA}$ and $F \in \mathcal{C}$ we have $f(F[V]) \leq f(F)$.
4. \mathfrak{A} for input F either returns a satisfying partial assignment φ_F for F or a non-empty set of variables $\emptyset \neq V_F$ used in some resolution refutation F .

Then for input $F_0 \in \mathcal{C}$ we can compute the lean kernel as well as an autarky realising the lean kernel (a “quasi-maximal autarky”) in time $\tilde{O}(f(F_0))$ as follows:

- Let $F := F_0$.
- Run \mathfrak{A} on F . If the output is a satisfying assignment φ , then return (F, φ) , while if the output is a set V of variables then let $F := F[V]$ and repeat.

For the pair (F, φ) computed we have $F = N_a(F_0)$, $\varphi \in \text{Auk}(F_0)$ and $\varphi * F_0 = F$.

Given a quasi-maximal autarky, we can determine the variables which are in the lean kernel but not in the largest autark subset, and then by arbitrary extension we can also compute a maximal autarky. It is not too complicated to provide a backtracking SAT solver (look-ahead or conflict-driven) without much space- or time-overhead with the ability to output in case of unsatisfiability the set of variables involved in the resolution refutation found; note that we do not need the resolution refutation itself, which can require exponential space, but only the set of variables involved, computed recursively bottom-up from the leaves, cutting off branches if the branching variable is not used as resolution variable.⁷

⁷This realises “intelligent backtracking” also for conflict-driven solvers, and is implemented in the `OKsolver-2002`.

For example, we can use $\mathcal{C} = \mathcal{CLS}$, $f = b^{n(F)}$ where b depends on the maximal clause-length of F , and for \mathfrak{A} the classical CNF algorithm from [MS85, Luc84] (branching on a shortest clauses, and using the autarky found when no clause is shortened; see Subsection 14.10.5), and we obtain:

Corollary 14.10.2. *For a clause-set $F \in \mathcal{CLS}$ let $\text{rk}(F) \in \mathbb{N}_0$ denote the maximal clause-length. Then a maximal autarky for input $F \in \mathcal{CLS}$ can be computed in time $\tilde{O}(\tau_{\text{rk}(F)-1}^{n(F)})$, where $\tau_{\text{rk}(F)-1} = \tau(1, \dots, \text{rk}(F) - 1)$ is the solution $x > 1$ of $\sum_{i=1}^{\text{rk}(F)-1} x^{-i} = 1$ (see Subsection 8.3.2 in Chapter 8 on branching heuristics). For inputs restricted to 3-CNF we have $\text{rk}(F) \leq 3$, where $\tau(1, 2) = 1.618\dots$, that is, a maximal autarky for inputs F in 3-CNF can be computed in time $O(1.619^{n(F)})$.*

It is not known to what extent the improved k -CNF bounds can be applied: Closest is [Kul99b], but as shown in [Kul99c], the use of blocked clauses cannot be simulated by resolution, so the meta-theorem is not applicable (at least *prima facie*). All other algorithms are based on some form of probabilistic reasoning, and thus in case of unsatisfiability do not yield (in some form) resolution refutations, so also here the meta-theorem (Theorem 14.10.1) is not applicable.

An application of the computation of the lean kernel to the problem of finding “unsatisfiable cores” in clause-sets is discussed in [KLMS06] (concentrating especially on the different roles of clauses, like “necessary”, “potentially necessary”, “usable” or “unusable” clauses).

14.10.4. Reduction to SAT

A translation of the autarky search problem to SAT has been introduced in [LS08], considering applications similar to those mentioned in [KLMS06]; the basic method used for the translation is the use of “clause indicator variables” for clauses satisfied by the autarky, and using also “variable indicator variables” to indicate whether an original variable is used by the autarky. The experimental results suggest that this works complementary to the method used in [KLMS06] (see the previous Subsection 14.10.3): Computation of the lean kernel via the autarky-resolution duality, based on a conflict-driven solver here, tends to use many iterations, while the algorithm of [LS08] searches directly for the largest autark subset (to express this, the clause-indicator variables are used, together with “AtMost”-constraints).

There are many variations possible for such translations, but the basic translation, encoding autarkies for clause-set F exactly as satisfying assignments of $A(F)$, is simple: For every variable $v \in \text{var}(F)$ introduce three variables $a_{v,\varepsilon}$ for $\varepsilon \in \{0, 1, *\}$, where “*” stands for “unassigned”, and $a_{v,\varepsilon}$ means that v shall get “value” ε ; so we have $n(A(F)) = 3n(F)$. A literal x over $\text{var}(F)$ is interpreted by the positive literal $a(x)$ over $\text{var}(A(F))$, where for positive literals v we set $a(v) := a_{v,1}$, while for negative literals \bar{v} we set $a(\bar{v}) := a_{v,0}$. Now a clause $C \in F$ yields $|C|$ many clauses of length $|C|$, namely for each $x \in C$ the “autarky clause” $C_x := \{\bar{a}(\bar{x})\} \cup \{a(y) : y \in C \setminus \{x\}\}$ (which requires that if one literal x is set to false, then some other literal y must be set to true). Then $A(F)$ consists of the clauses C_x for $C \in F$ and $x \in C$ together with the ALO-clauses, that is, for every

variable $v \in \text{var}(F)$ the (positive, ternary) clause $\{a_{v,0}, a_{v,1}, a_{v,*}\}$, and together with the AMO-clauses, that is for each $v \in \text{var}(F)$ and $\varepsilon, \varepsilon' \in \{0, 1, *\}$, $\varepsilon \neq \varepsilon'$ the (negative, binary) clause $\{\overline{a_{v,\varepsilon}}, \overline{a_{v,\varepsilon'}}\}$. Obviously we have a natural bijection from $\text{mod}(A(F))$ to $\text{Auk}^r(F)$ (note that every satisfying (partial) assignment for $A(F)$ must actually be total). A problem now is that to express non-triviality of the autarky, we need the long clause $\{\overline{a_{v,*}} : v \in \text{var}(F)\}$ (containing *all* variables); to avoid this clause of maximal length, and to obtain more control over the autarky obtained, [LS08] uses the above mentioned indicator variables.

14.10.5. Autarkies for worst-case upper bounds

Apparently the first application of “autarkies” (without the name, which was invented 10 years later) can be found in [EIS76], where they are used to show that 2-SAT is decidable in polynomial time. This very simple method works as follows for a clause-set F with maximal clause-length $\text{rk}(F) \leq 2$: Consider any literal x in F , and let the partial assignment φ_x be obtained from $\langle x \rightarrow 1 \rangle$ by adding all inferred unit-clause propagations for F . If $\perp \in \varphi_x * F$, then F is satisfiability-equivalent to $\langle x \rightarrow 0 \rangle * F$ and we eliminated one variable. Otherwise we know that all clauses in $\varphi_x * F$ must have length at least 2, and thus, due to $\text{rk}(F) \leq 2$, φ_x is an autarky for F — so in this case F is satisfiability equivalent to $\varphi_x * F$, and again at least one variable has been eliminated.

If F has unrestricted clause-length, then in case of $\perp \notin \varphi_x * F$ and where furthermore φ_x is not an autarky for F (which can be easily checked), we know that F must contain a clause C of length $2 \leq |C| \leq \text{rk}(F) - 1$ (note that this generalises the above argument). Applying this argument to both branches $\varphi_x, \varphi_{\bar{x}}$ has been exploited in [MS85] for improved k -SAT upper bounds (and in this article also the notion of an “autark assignment” has been introduced). Basically the same argument, but in a different form, has been used in [Luc84]; see [KL97, KL98] for discussions (the point is while [MS85] checks whether $\varphi_x, \varphi_{\bar{x}}$ are autarkies for the current “residual” clause-set (at each node of the search tree), [Luc84] only checks the special cases which are actually needed to prove the bound).

These ideas have been systematically extended in [Kul99b], on the one hand generalising autarkies to “Br-autarkies”, allowing for the application of an arbitrary reduction and the possible elimination of blocked clauses afterwards. And on the other hand, the notion of a *weak k -autarky* is introduced (where “weak” is added now according to the general distinction between “weak autarkies” and “autarkies”) for $k \geq 0$, which is a partial assignment φ such that $|(\varphi * F) \setminus F| \leq k$; note that weak 0-autarkies are exactly the weak autarkies. For $k \geq 1$ in general $\varphi * F$ is no longer satisfiability-equivalent to F , however the point is that, using $N := (\varphi * F) \setminus F$ for the set of new clauses created by φ , for every partial assignment ψ with $\text{var}(\psi) \subseteq \text{var}(N)$ (thus $\text{var}(\psi) \cap \text{var}(\varphi) = \emptyset$) and $\psi * N = \top$ (i.e., ψ satisfies N) the partial assignment φ is a weak autarky for $\psi * F$.⁸ In [Kul99b] this k -autarky principle has been applied by considering a complete branching (ψ_1, \dots, ψ_m) for N (i.e., covering all total assignments), where then those branches ψ_i satisfying N can be augmented by φ . In Subsection 14.13.2

⁸The proof is: $(\varphi * (\psi * F)) \setminus (\psi * F) = ((\varphi \circ \psi) * F) \setminus (\psi * F) = ((\psi \circ \varphi) * F) \setminus (\psi * F) = (\psi * (\varphi * F)) \setminus (\psi * F) \subseteq (\psi * (N \cup F)) \setminus (\psi * F) = ((\psi * N) \cup (\psi * F)) \setminus (\psi * F) = \top$.

we discuss another possibility to exploit this situation, by adding “look-ahead autarky clauses”.

Finally we mention the “black and white literals principle” ([Hir00]), which generalises the “generalised sign principle” from [KL97], and which reformulates the “autarky principle” (that applying an autarky yields a satisfiability-equivalent clause-set) in such a way that it becomes easily applicable as a source for reductions in worst-case upper-bound proofs (see [KL98] for a discussion).

14.10.6. Incomplete autarky search in complete SAT solvers

The basic idea from [MS85], as explained in Subsection 14.10.5, is to check the assignments φ_x (obtained by adding all inferred unit-clause propagation to a test-assignment $x \rightarrow 1$) for being an autarky (in which case there is no need to branch). This has been implemented in the `OKsolver-2002` (see [Kul98, Kul02])⁹, incorporating the ideas from [Kul99b]:

- Now arbitrary binary splitting are considered (and not clause-branching as in [MS85]).
- The assignments φ_x occur naturally when performing “failed-literal reduction”.
- The branching heuristics is based on counting new clauses, and the autarky-test is actually necessary to avoid branches without “progress” (i.e., with zero new clauses).

See Subsection 8.7.4.1 in Chapter 8 on the general theory of branching heuristics for more information. This type of “ad-hoc” and “enforced” autarky search has also been applied (with variations) in the `march` solvers.

14.10.7. Employing autarkies in generalised unit-clause-propagation

Unit-clause propagation can be strengthened to failed-literal reduction, and further to arbitrary levels (corresponding to levels of generalised Horn formulas); see [Kul99a] for a thorough treatment and an overview on the literature. There, through the use of oracles, special poly-time procedures for finding autarkies (e.g., finding “linear autarkies”; see Subsection 14.10.9) are employed, enhancing the hierarchies by adding more satisfiable instances to each level.

This approach only uses autarkies at the leaves (to ensure confluence); in a more ad-hoc manner in [DE92] the assignments found when searching for enforced assignments at the various levels of the hierarchy (generalising the approach from Subsection 14.10.6 in a certain way) are directly checked for the autarky property.

14.10.8. Local search

Local search solvers search through total assignments, and each such total assignment can be inspected for being an autarky, according to Lemma 14.9.1. A first implementation one finds in `UnitMarch` (enhancing the local search solver `UnitWalk`; see [Hv08]). Combinations with other local search solvers might become

⁹this “historic” solver is now part of the `OKlibrary` (<http://www.ok-sat-library.org>)

interesting in the future (especially when adapting the heuristics to the new goal of finding autarkies). Direct local search through partial assignments should also be interesting, but has not been tried yet.

14.10.9. Polynomial-time cases

Most classes of clause-sets, where basic tasks like autarky finding or computation of the lean kernel can be performed in polynomial time, are handled by special types of autarkies, which is the subject of the following Section 14.11. Here we only comment on the most prominent cases, and especially how they can be handled by the techniques discussed in the current section (considering general autarkies); see [KMT08] for a more elaborated study.

We already mentioned in Subsection 14.10.5 that the consideration of clause-sets F in 2-CNF (every clause contains at most two literals) was at the beginning of “autarky theory”. And it is easy to see that actually if F is not lean then there must be some literal x such that φ_x (which was the extension of $\langle x \rightarrow 1 \rangle$ by unit-clause propagations) is an autarky (see [KMT08] for more details). So the autarky-existence problem is in P for 2-CNF, and (since 2-CNF is stable under the application of partial assignments) also the lean kernel is computable in polynomial time. This method for finding autarkies is covered by the method discussed in Subsection 14.10.6 for finding “ad-hoc enforced autarkies”, and thus is actually implemented in look-ahead solvers like the `OKsolver-2002` or the `march-solvers` (see Subsection 14.10.6). Another way to handle 2-CNF is by Theorem 14.10.1, where for finding a resolution-refutation one could also apply the stability of 2-CNF under the resolution rule (though finding a resolution refutation by a backtracking solver is more efficient). Finally every autarky for F in 2-CNF is a special case of a “linear autarky”, as discussed in Subsection 14.11.3, and thus can be handled also by the special methods there.

In a certain sense “dual” to 2-CNF is the class of clause-sets F such that every variable occurs at most once positively and at most once negatively. Minimally unsatisfiable such F are exactly the marginal minimally unsatisfiable clause-sets of deficiency 1. It is well known (and trivial) that by DP-resolution (un)satisfiability for this class is decidable in polynomial time, so we can find easily a resolution refutation, while by using self-reduction we can also find a satisfying assignment; so again by Theorem 14.10.1 we obtain that the computation of the lean kernel (and the computation of a maximal autarky) can be done in polynomial time. Another approach is given by the observation that every autarky for such F must be a “matching autarky”, as discussed in Subsection 14.11.2, and thus we can use the (poly-time) methods from there. Finally, minimal autarkies of F correspond to the cycles in the conflict multigraph of F (compare [Kul04a] for the notion of the conflict (multi-)graph), and thus we can find them by finding cycles.

Finally we mention that also for Horn clause-sets we can compute the lean kernel in polynomial time by Theorem 14.10.1. Again this can be handled also by linear autarkies (see Subsection 14.11.3).

14.10.10. Final remarks on the use of autarkies for SAT solving

Until now, the applications of autarky search to practical SAT solving have been rather restricted, mostly focusing on searching autarkies directly in the input, while autarky search at each node of the search tree, as discussed in Subsection 14.10.6, just checks whether “by chance” the partial assignments at hand are actually autarkies. Now in special cases autarky search as preprocessing will help, but stronger results can only be expected when *systematically* searching for autarkies at *each* node of the search tree. We close this section by showing (by example) that under “normal circumstances” relevant autarkies are created deeper in the search tree. Consider an input clause-set F , and we ask how many autarkies we obtain after application of a partial assignment φ . To simplify the discussion, we assume that F is unsatisfiable (these are also the relevant hard cases created at least during search). The basic intuition is that we obtain the more autarkies the more “redundancies” are contained in F . By definition an unsatisfiable clause-set is “irredundant” iff it is minimally unsatisfiable, and minimally unsatisfiable clause-sets are lean. Now there are stronger “degrees k of irredundancy”, achieved by requiring that for every partial assignment φ with $n(\varphi) \leq k$ for some given k we have that $\varphi * F$ is minimally unsatisfiable. Minimally unsatisfiable F are characterised by $k = 0$, saturated minimally unsatisfiable F by $k = 1$, and unsatisfiable hitting clause-sets F by $k = \infty$; see Chapter 6 in [Kul07a] for investigations of these issues. So from the autarky-perspective unsatisfiable hitting clause-sets are the most intractable cases — whatever partial assignment is applied, never any autarky arises. On the other end of the spectrum we have marginal minimally unsatisfiable clause-sets F of deficiency 1 — here already for $n(\varphi) = 1$ the clause-set $\varphi * F$ decomposes in variable-disjoint components, one of them minimally unsatisfiable (again marginal minimally unsatisfiable of deficiency 1), while all other components are satisfiable (and actually “matching-satisfiable”; see Subsection 14.11.2), and so we have (easy, relevant) autarkies. For practical applications, instances will not even be minimally unsatisfiable, and typically also are assembled from several parts, and thus the appearance of relevant autarkies not too deep down the search tree is to be expected.

14.11. Autarky systems: Using weaker forms of autarkies

In this section we consider restricted notions of autarkies, which allow to find such restricted autarkies in polynomial time, while still sharing many of the properties with general autarkies. The first example for a special form of autarkies, “linear autarkies”, was introduced in [Kul00b], and the general properties of the formation $F \mapsto \text{Auk}(F)$ of the (full) autarky monoid have been paralleled with the properties of the sub-monoid $F \mapsto \text{LAuk}(F) \leq \text{Auk}(F)$ of linear autarkies. Based on these investigations, in [Kul03] various forms of “autarky systems” have been introduced, axiomatising the basic properties. Some preliminary study on further generalisations can be found in [Kul01], while the notions have been revised and adapted to clause-sets with non-boolean variables in [Kul07a].

In the light of Lemma 14.9.3, an appropriate definition of an “autarky system” would be that of a contravariant functor $\mathcal{A} : \mathcal{CLS} \rightarrow \mathfrak{MN}$ such that the

canonical inclusion (of sub-monoids) is a natural transformation $\mathcal{A} \rightarrow \text{Auk}$. But since the study of such categorical notions in the context of SAT has not really started yet, we refrain here from such functorial formulations, and call an **autarky system** a map \mathcal{A} defined on all clause-sets such that $\mathcal{A}(F)$ is a sub-monoid of $\text{Auk}(F)$ for all $F \in \mathcal{CLS}$, and such that for $F_1 \subseteq F_2$ we have $\mathcal{A}(F_2) \subseteq \mathcal{A}(F_1)$. The most basic examples of autarky systems are $\mathcal{A} = \text{Auk}$, the full autarky system, and $\mathcal{A} = (\{\langle \rangle\})_{F \in \mathcal{CLS}}$, the trivial autarky system. From an autarky system \mathcal{A} we obtain the *restricted system* $\mathcal{A}^r(F) := \mathcal{A}(F) \cap \text{Auk}^r(F)$, which is again an autarky system if \mathcal{A} is “standardised” (that is, variables not in F do not matter; see below for the definition). The restricted version is especially important for practical considerations, however for theoretical investigations mostly the “full” versions (also using variables not in F) are more convenient. As before, we call the minimal elements of $\mathcal{A}^r(F) \setminus \{\perp\}$ *minimal \mathcal{A} -autarkies*, and the maximal elements of $\mathcal{A}^r(F)$ *maximal \mathcal{A} -autarkies*. If \mathcal{A}^r is an autarky system, then all maximal autarkies have the same domain (use the same variables).

A clause-set F is called *\mathcal{A} -lean* if $\mathcal{A}(F)$ contains only trivial autarkies. Using \mathcal{A} -autarky reduction instead of autarky reduction, i.e., reduction $F \rightsquigarrow \varphi * F$ for $\varphi \in \mathcal{A}(F)$, generalising Subsection 14.8.3 we obtain the *\mathcal{A} -lean kernel* $N_{\mathcal{A}}(F) \subseteq F$, the largest \mathcal{A} -lean sub-clause-set of F ; note that due to the anti-monotonicity condition for an autarky system, \mathcal{A} -autarky reduction still is confluent. Again we have that $N_{\mathcal{A}}(F) = F$ holds iff F is \mathcal{A} -lean, while in case of $N_{\mathcal{A}}(F) = \top$ we call F *\mathcal{A} -satisfiable*. The lean kernel formation $F \mapsto N_{\mathcal{A}}(F)$ is again a kernel operator. The union of \mathcal{A} -lean clause-sets is \mathcal{A} -lean, and $N_{\mathcal{A}}(F)$ is the union of all \mathcal{A} -lean sub-clause-sets of F .

If there exists $\varphi \in \mathcal{A}(F)$ with $\varphi * F = \top$, then F is \mathcal{A} -satisfiable, but the reverse direction does not hold in general. In other words, using *\mathcal{A} -autark subsets* of F as sub-clause-sets satisfied by some \mathcal{A} -autarky for F , we again have that the union of \mathcal{A} -autark subsets is again an \mathcal{A} -autark subsets, and thus there is a largest \mathcal{A} -autark subsets, where the largest \mathcal{A} -autark subset is disjoint with the largest lean subset $N_{\mathcal{A}}(F)$ — however we do not obtain a partition of F in this way, and additional properties of autarky systems are needed. We do not want to discuss various forms of strengthened autarky systems, but we only consider the strongest form, called a **normal autarky system**, which comprises the six basic properties of autarkies listed in Subsection 14.8.2. More precisely, we require the following five additional properties (for all partial assignments φ, ψ , all clause-sets F and all finite sets V of variables):

standardised $\varphi \in \mathcal{A}(F) \Leftrightarrow \varphi|_{\text{var}(F)} \in \mathcal{A}(F)$;

iterative if $\varphi \in \mathcal{A}(F)$ and $\psi \in \mathcal{A}(\varphi * F)$, then $\psi \circ \varphi \in \mathcal{A}(F)$;

invariant under variable elimination if $\text{var}(\varphi) \cap V = \emptyset$, then $\varphi \in \mathcal{A}(F) \Leftrightarrow \varphi \in \mathcal{A}(V * F)$;

\perp -invariant $\mathcal{A}(F) = \mathcal{A}(F \cup \{\perp\})$;

invariant under renaming if F' is isomorphic to F , then the same isomorphism turns $\text{Auk}(F)$ into $\text{Auk}(F')$.

(Note that for the functorial definition of an autarky system mentioned above, the property of invariance under renaming is automatically fulfilled.) If \mathcal{A} is a normal autarky system, then the restricted system \mathcal{A}^r is an autarky system which fulfils

all additional properties except of being standardised (that is, \mathcal{A}^F is iterative, invariant under variable elimination, \perp -invariant and invariant under renaming). If $\mathcal{A}_1, \mathcal{A}_2$ are (normal) autarky systems, then also $F \mapsto \mathcal{A}_1(F) \cap \mathcal{A}_2(F)$ is a (normal) autarky system.¹⁰

We assume from now on that \mathcal{A} is a normal autarky system, which can be justified by the fact that every autarky system has a unique strengthening which is a normal autarky system (just by adding the “missing” autarkies). Now for all clause-sets F there is an \mathcal{A} -maximal autarky φ with $\varphi * F = N_{\mathcal{A}}(F)$, especially F is \mathcal{A} -satisfiable iff there is $\varphi \in \mathcal{A}(F)$ with $\varphi * F = \top$. Regarding the full autarky system, from an autarky φ with $\varphi * F = N_{\mathcal{A}}(F)$ we can easily obtain a maximal autarky φ' , by adding arbitrary assignments for variables in $\text{var}(F \setminus \varphi * F)$ which do not already have a value (and also variables not in F need to be removed). However for arbitrary (normal) autarky systems this might not be possible, and so we call \mathcal{A} -autarkies φ with $\varphi * F = N_{\mathcal{A}}(F)$ *quasi-maximal*. Every \mathcal{A} -maximal autarky is \mathcal{A} -quasi-maximal.

If F is \mathcal{A} -lean, then also $V * F$ and $F[V]$ are \mathcal{A} -lean for (finite) sets V of variables, and every clause-set $F' \supseteq F$ with $\text{var}(F') = \text{var}(F)$ is \mathcal{A} -lean as well.¹¹ The \mathcal{A} -autark subsets of F , i.e., the subsets $F_{\text{var}(\varphi)}$ for $\varphi \in \mathcal{A}(F)$, can be characterised as the \mathcal{A} -satisfiable subsets F_V for $V \subseteq \text{var}(F)$, where furthermore F_V is \mathcal{A} -satisfiable iff $F[V]$ is \mathcal{A} -satisfiable. The set of \mathcal{A} -autark subsets of F is closed under union, with smallest element \top and largest element $F \setminus N_{\mathcal{A}}(F)$. The partition of F into $N_{\mathcal{A}}(F)$ and $F \setminus N_{\mathcal{A}}(F)$ (where some part can be empty) has, as before, several characterisations; it is the partition of F into the largest \mathcal{A} -lean and the largest \mathcal{A} -autark subset, or, equivalently, if $F = F_1 \cup F_2$, where F_1 is \mathcal{A} -lean and $\text{var}(F_1) * F_2$ is \mathcal{A} -satisfiable, then $F_1 = N_{\mathcal{A}}(F)$ and $F_2 = F \setminus N_{\mathcal{A}}(F)$ (and also the other way around; note that disjointness of F_1, F_2 actually follows from $\text{var}(F_1) * F_2$ being satisfiable). \mathcal{A} -satisfiability is “self-reducible” in the sense that (in the same way as for the full autarky system) from an oracle deciding \mathcal{A} -leanness we obtain efficiently a way for computing the lean kernel (see Lemma 8.6 in [Kul03]); however for also computing an \mathcal{A} -quasi-maximal autarky (as in [KMT08]) additional conditions on the autarky system \mathcal{A} are needed.

14.11.1. Pure autarkies

In Example 14.8.3 we have already seen that every pure literal x for a clause-set F induces an autarky $\langle x \rightarrow 1 \rangle \in \text{Auk}(F)$. This is not yet an autarky system, since we do not have closure under composition. So we call a partial assignment φ a *simple pure autarky* for F if for all literals x with $\varphi(x) = 1$ we have $\bar{x} \notin \bigcup F$; the set of all simple pure autarkies is denoted by $\text{PAut}_0(F)$. In this way we obtain an autarky system $F \mapsto \text{PAut}_0(F)$, which also is standardised, invariant

¹⁰More generally, arbitrary intersections of (normal) autarky systems are again (normal) autarky systems, and we obtain the complete lattice of autarky systems and the complete lattice of normal autarky systems, and the corresponding closures (hulls; as always in algebra).

¹¹We remark here that for *formal clause-sets*, which correspond to the notion of a hypergraph and are pairs (V, F) for sets V of variables and clause-sets over V (so that now “formal variables” are allowed, i.e., elements of $V \setminus \text{var}(F)$), \mathcal{A} -leanness of (V, F) implies $V = \text{var}(F)$, since otherwise a non-trivial autarky would exist.

under variable elimination, \perp -invariant, and invariant under renaming, however not iterative (applying a simple pure autarky can create new pure literals).

Example 14.11.1. For $F := \{\{a, b\}, \{\bar{b}\}\}$ we have $\text{PAut}_0(F) = \{\langle \rangle, \langle a \rightarrow 1 \rangle\}$, and so the unique PAut_0 -maximal autarky is $\langle a \rightarrow 1 \rangle$. However this is not a satisfying assignment for F , despite the fact that F is PAut_0 -satisfiable.

To obtain a normal autarky system, according to the general theory we consider $\text{PAut}(F)$, which is the closure of $\text{PAut}_0(F)$ under “iteration” of autarkies. So the elements of $\text{PAut}(F)$ are the partial assignments $\varphi_m \circ \dots \circ \varphi_1$, where φ_i is a simple pure autarky for F_{i-1} , where $F_0 := F$ and $F_{i+1} := \varphi_{i+1} * F_i$. The elements of $\text{PAut}(F)$ are called *pure autarkies*.

Given any autarky system \mathcal{A} , it is natural to study which random clause-sets (say, in the constant-density model, fixing a clause-length $k \geq 2$, and considering all k -uniform clause-lists of length c over n given variables as equally likely) are \mathcal{A} -satisfiable. Experimental evidence seems to suggest that at least the “natural” autarky systems considered here all show a threshold behaviour (below a certain threshold density every clause-list is almost surely \mathcal{A} -satisfiable, above that density almost surely none). Only for pure autarkies this has been finally rigorously established in [Mol05], proving that for all $k \geq 2$ a threshold exists, at the density $\frac{c}{n} = 1$ for $k = 2$ and at $\frac{c}{n} = 1.636\dots$ for $k = 3$.

14.11.2. Matching autarkies

The notion of “matching autarkies” is based on the work in [FV03], which introduced “matched clause-sets”, that is, clause-sets F with $d^*(F) = 0$ (recall Definition 14.2.1¹²), which are the *matching satisfiable clause-sets* using our systematic terminology for autarky systems (that is, clause-sets satisfiable by matching autarkies). Perhaps the earliest work where matching autarkies are implicitly used is [AL86]. Matching satisfiable sub-clause-sets of a clause-set F as the independent sets of a transversal matroid have been used in [Kul00a] to show that SAT for clause-sets with bounded maximal deficiency can be decided in polynomial time. The first thorough study of matching lean clause-sets (clause-sets without non-trivial matching autarkies) one finds in [Kul03], completed (and extended to clause-sets with non-boolean variables) in [Kul07a]. As shown there, the basic tasks like finding a maximal matching autarkies can be performed in polynomial time. The structure of the matching autarky monoid has not been investigated yet.

As introduced in [Kul03], “matching autarkies” are partial assignments which satisfy in every clause they touch a literal with underlying *unique* variable.

Definition 14.11.1. A partial assignment φ is called a *matching autarky* for a clause-set F if there is an injection $v : F_{\text{var}(\varphi)} \rightarrow \text{var}(\varphi)$ such that for all clauses $C \in F_{\text{var}(\varphi)}$ there is a literal $x \in C$ with $v(C) = \text{var}(x)$ and $\varphi(x) = 1$.

Clearly every matching autarky is in fact an autarky. The set of matching autarkies is denoted by $\text{MAuk}(F)$ resp. $\text{MAuk}^r(F)$ (considering only matching autarkies with variables in F).

¹²we remark that also the notion of “deficiency” was introduced in [FV03]

Example 14.11.2. If every literal occurs only once, then every autarky is a matching autarky; see for example Example 14.9.1. As the simplest example where some autarkies are left out consider $F := \{\{v_1\}, \{v_1, v_2\}\}$; we have $\text{MAuk}^r(F) = \{\langle \rangle, \langle v_2 \rightarrow 1 \rangle, \langle v_1 \rightarrow 1, v_2 \rightarrow 1 \rangle\}$ (note that $\langle v_1 \rightarrow 1 \rangle$ is an autarky for F but not a matching autarky). While F is still matching satisfiable, the clause-set $\{\{v_1, v_2\}, \{v_1, \bar{v}_2\}, \{\bar{v}_1, v_2\}\}$ is satisfiable but matching lean.

Using *multi*-clause-sets F instead of clause-sets, a partial assignment φ is a matching autarky for F iff $F[\text{var}(\varphi)]$ is matching satisfiable (we have to use multi-clause-sets in order to avoid the contraction of clauses).

Example 14.11.3. $\langle v \rightarrow 1 \rangle$ is *not* a matching autarky for $F := \{\{v\}, \{w\}, \{v, w\}\}$, and accordingly as multi-clause-set $F[\text{var}(\varphi)]$ consists of two occurrences of the clause $\{v\}$, and thus is not matching satisfiable. But as a clause-set we would obtain $F[\text{var}(\varphi)] = \{\{v\}\}$ which is matching satisfiable.

The operator $F \mapsto \text{MAuk}(F)$ yields a normal autarky system, but only for multi-clause-sets; for clause-sets we have an autarky system which fulfils all properties of normal autarky systems except of being invariant under variable elimination. The matching satisfiable clause-sets are exactly the clause-sets with $d^*(F) = 0$, while the matching lean clause-sets are exactly the clause-sets F with $\forall F' \subset F : d(F') < d(F)$ (thus $d^*(F) = d(F)$, and if $F \neq \top$, the $d(F) \geq 1$). This generalises the fact from [AL86] that minimally unsatisfiable clause-sets have deficiency at least 1. More on matching autarkies one finds in Section 7 of [Kul03], and in Section 4 of [Kul07a].

14.11.3. Linear autarkies

The notion of a “linear autarky”, introduced in [Kul00b], was motivated by the applications of linear programming to SAT solving in [vW00] (see Theorem 2 there). The basic notion is that of “simple linear autarkies”, yielding an autarky system which fulfils all conditions of a normal autarky system except of being iterative, which then is repaired by the notion of a “linear autarky”. Let a *variable weighting* be a map $w : \mathcal{VA} \rightarrow \mathbb{Q}_{>0}$, that is, every variable gets assigned a positive rational number (real numbers could also be used). We extend the weighting to literals and clauses, relative to a partial assignment φ , as follows (using an arithmetisation of “false” by -1 and “true” by $+1$):

- For a literal x let $\text{sgn}_\varphi(x) := 0$ if $\text{var}(x) \notin \text{var}(\varphi)$, otherwise $\text{sgn}_\varphi(x) := +1$ if $\varphi(x) = 1$, while $\text{sgn}_\varphi(x) := -1$ if $\varphi(x) = 0$.
- Now for a literal x let $w_\varphi(x) := \text{sgn}_\varphi(x) \cdot w(\text{var}(x))$.
- And for a clause C we define $w_\varphi(C) := \sum_{x \in C} w_\varphi(x)$,

Note that for clauses C we have $w_\varphi(\bar{C}) = -w_\varphi(C)$. Now we can state the definition of “simple linear autarkies” (as introduced in [Kul00b]):

Definition 14.11.2. A partial assignment φ is a *simple linear autarky* for a clause-set F if there exists a weighting $w : \mathcal{VA} \rightarrow \mathbb{Q}_{>0}$ such that for all $C \in F$ we have $w_\varphi(C) \geq 0$.

Clearly a simple linear autarky is an autarky, and also clearly we only need to consider the variables actually occurring in F . Defining *linear autarkies* as the closure of simple linear autarkies under iteration we obtain a normal autarky system.

14.11.3.1. Comparison to pure and matching autarkies

Example 14.11.4. Every pure literal x for a clause-set F yields a simple linear autarky $\langle x \rightarrow 1 \rangle$, certified by any constant weighting. The partial assignment $\langle v_1 \rightarrow 1, v_2 \rightarrow 2 \rangle$ is a (satisfying) simple linear autarky for the matching lean clause-set $\{\{v_1, v_2\}, \{v_1, \bar{v}_2\}, \{\bar{v}_1, v_2\}\}$ from Example 14.11.2, again certified by any constant weighting. The partial assignment $\varphi := \langle v_1, v_2, v_3 \rightarrow 1 \rangle$ is not a simple linear autarky for $F := \{\{v_1, \bar{v}_2, \bar{v}_3\}, \{\bar{v}_1, v_2\}, \{v_3\}\}$, since if it had a certificate w , then $w(\bar{v}_3) < 0$ in the first clause, so from $0 \leq w(\{v_1, \bar{v}_2, \bar{v}_3\}) = w(\{v_1, \bar{v}_2\}) + w(\bar{v}_3)$ we obtain $w(\{v_1, \bar{v}_2\}) \geq -w(\bar{v}_3) > 0$, but $w(\{\bar{v}_1, v_2\}) = -w(\{v_1, \bar{v}_2\})$, and thus the second clause gets a negative weight. On the other hand, φ is a matching autarky for F .

So it might seem that simple linear autarkies and matching autarkies are incomparable in strength, however this is misleading: every clause-set which has a non-trivial matching autarky also has a non-trivial simple linear autarky, as shown in [Kul00b] (by basic linear algebra one shows that a matching satisfiable clause-set $F \neq \top$ must have a non-trivial simple linear autarky, and thus F must be linearly satisfiable (by iteration); using invariance under variable elimination this is lifted to arbitrary non-matching-lean F). So the linearly-lean kernel of a clause-set is always contained in the matching-lean kernel. In Example 14.11.4 the clause-set F has the simple linear autarky $\langle v_1 \rightarrow 1, v_2 \rightarrow 1 \rangle$, and while the partial assignment φ is not a simple linear autarky, it is actually a linear autarky.

Example 14.11.5. Consider $F := \{\{v_1, \bar{v}_2, \bar{v}_3\}, \{\bar{v}_1, v_2, \bar{v}_3\}, \{\bar{v}_1, \bar{v}_2, v_3\}\}$. The partial assignment $\varphi := \langle v_1, v_2, v_3 \rightarrow 1 \rangle$ is a matching autarky for F but not a simple linear autarky; furthermore φ is a minimal autarky for F and thus φ is also not a linear autarky. So we see that, although the reduction power of linear autarkies is as least as big as the reduction power of matching autarkies (in this example, $\langle v_1, v_2 \rightarrow 0 \rangle$ is a simple linear autarky for F), there are matching autarkies which do not contain any non-trivial linear autarkies.

14.11.3.2. Matrix representations of clause-sets

Now we turn to the question of finding linear autarkies in polynomial time. This is best done by translating the problem into the language of linear programming. For a clause-set F consider the clause-variables matrix $M(F)$ of size $c(F) \times n(F)$. Let us emphasise that this is different to the use of matrices in Example 14.2.1, where matrices only are used for an economic representation, while here now “mathematical” matrices are used, with entries from $\{-1, 0, +1\} \subset \mathbb{Z}$, where the rows correspond to the clauses of F (in some order), and the columns correspond to the variables of F (in some order). Now F is linearly lean if and only if $M(F) \cdot \vec{x} \geq 0$ has the only solution $\vec{x} = 0$, and furthermore every solution \vec{x} corresponds to a simple linear autarky by the same arithmetisation as used before,

i.e., 0 corresponds to “not assigned”, +1 corresponds to “true” and -1 corresponds to “false”. Thus by linear programming we can find a non-trivial linear autarky (if one exists).

Via linear autarkies, several basic cases of poly-time SAT decision are covered. Using a constant weighting, we see that every autarky for a clause-set F in 2-CNF (every clause has length at most 2) is a simple linear autarky, and thus every satisfiable 2-CNF is linearly satisfiable. [van00] applies linear autarkies to q -Horn formulas, while in [Kul99a] it is shown how satisfiable generalised Horn clause-sets (in a natural hierarchy of generalised Horn clause-sets, covering all clause-sets) can be captured by linearly satisfiable clause-sets (used as an oracle to amplify the general hierarchy studied in [Kul99a]). For more on linear autarkies see [Kul00b, Kul03], while [Kul06] contains more on “balanced” linear autarkies (see Subsection 14.11.4). In general the field of linear autarkies appears still largely unexplored; for example there seems to be a threshold for linear satisfiability of random 3-CNF at around density 2 (i.e., having twice as many clauses as variables).

Since matrix representations of satisfiability problems are at least of great theoretical value, we conclude this subsection on linear autarkies by some general comments on technical problems arising in this context. First, what is a “matrix”?! The appropriate definition of the notion of a (general) matrix is given in [Bou89], where a *matrix over a set V* is defined as a triple (R, C, f) , where R, C are arbitrary sets (the “row indices” and the “column indices”), and $f : R \times C \rightarrow V$ gives the entries of the matrix. It is important to note that for row and column indices arbitrary objects can be used, and that no order is stipulated on them. In this context it is appropriate to introduce *labelled clause-sets* (corresponding to general hypergraphs) as triples (V, F_0, F) , where V, F_0 are arbitrary sets (the “variables” and the “clause-labels”), while $F : F_0 \rightarrow \mathcal{CLS}(V)$ maps every clause-label to a clause over V . Note that labelled clause-sets can have *formal variables*, i.e., variables $v \in V$ not occurring in the underlying clause-set ($v \notin \text{var}(F(F_0))$), and that clauses can occur multiple times (furthermore clauses have “names”). Now we have a perfect correspondence between finite matrices over $\{-1, 0, +1\}$ and finite labelled clause-sets, more precisely we have two ways to establish this correspondence, either using the *clause-variable matrix*, which assigns to (V, F_0, F) the matrix (F_0, V, F') (where $F'(C, v)$ is ± 1 or 0, depending on whether v occurs in $F(C)$ positively, negatively or not at all), or using the *variable-clause matrix* (V, F_0, F'') (with $F''(v, C) := F'(C, v)$). *Standardised matrices* use row-sets $R = \{1, \dots, n\}$ and column-sets $C = \{1, \dots, m\}$ for $n, m \in \mathbb{N}_0$, and in this way also “automatically” an order on the rows and columns is established (so that standardised matrices can be identified with “rectangular schemes of values”). Ironically this less appropriate notion of a matrix (with its ambiguous treatment of order) is common in the field of combinatorics, so that, in the spirit of [BR91], in this context we call the above matrices “combinatorial matrices”. In the context of the autarky part of this chapter, clause-variable matrices (as well as its transposed, variable-clause-matrices) are combinatorial matrices, so that we save us the ordeal of having to determine somehow an order on variables and clauses.¹³

¹³In Subsection 14.12.2 we will consider the determinant of (square) clause-variable matrices,

Clause-sets do not allow formal variables nor multiple clauses, and thus their clause-variable matrices don't have zero columns nor identical rows. Thus when we speak of a matrix corresponding to a clause-set, then by default we consider clause-variable matrices, and the matrices don't contain zero columns or identical rows (while when considering labelled clause-sets no such restrictions are applied).

14.11.4. Balanced autarkies

We now consider the important (normal) autarky system of “balanced autarkies” which, unlike pure, matching and linear autarkies, has an NP-complete search problem (as the full autarky system): the main motivation behind balanced autarkies perhaps can be understood as providing a bridge from hypergraph theory to satisfiability theory.

Definition 14.11.3. A partial assignment φ is a *balanced autarky* for a clause-set F if φ an autarky for F as well as for $\overline{F} = \{\overline{C} : C \in F\}$, or, in other words, if in every clause $C \in F$ touched by φ there are satisfied literals as well as falsified literals.

Some basic facts about balanced autarkies are as follows:

1. Balanced autarkies yield a normal autarky system.
2. Balanced-satisfiable clause-sets, i.e., clause-sets satisfiable by balanced autarkies, are exactly the instances of NAESAT (“not-all-equal-sat”), that is, clause-sets which have a satisfying assignment which in every clause also falsifies some literals.
3. φ is a balanced autarky for F iff φ is an autarky for $F \cup \overline{F}$.
4. For *complement-invariant* clause-sets, i.e., where $F = \overline{F}$ holds, the balanced autarkies are exactly the autarkies. Especially, the satisfying partial assignments of complement-invariant clause-sets are exactly the balanced-satisfying assignments.
5. A complement-invariant clause-set F is satisfiable iff the underlying variable-hypergraph $\mathfrak{H}(F) = (\text{var}(F), \{\text{var}(C) : C \in F\})$ is 2-colourable.¹⁴

An example for a class of balanced lean clause-sets (having no non-trivial balanced autarky) is the class of clause-sets $F \setminus \{C\}$ for $F \in \text{MU}(1)$ and $C \in F$ (note that every such $F \setminus \{C\}$ is matching satisfiable).¹⁵

and the determinant of a combinatorial matrix is only determined up to its sign — fortunately actually only the absolute value of the determinant is needed there!

¹⁴A *hypergraph* is a pair (V, E) , where V is the set of “vertices”, while E , the set of “hyperedges”, is a set of subsets of V . A 2-colouring of a hypergraph (V, E) is a map $f : V \rightarrow \{0, 1\}$ such that no hyperedge is “monochromatic”, i.e., the 2-colourings of F correspond exactly to the satisfying assignments for the (complement-invariant) clause-set $F_2((V, E)) := E \cup \overline{E}$ (using the vertices as variables).

¹⁵That every such $F \setminus \{C\}$ is balanced lean can be shown by induction on $n(F)$. The assertion is trivial for $n(F) = 0$, so assume $n(F) > 0$. F must contain a variable v occurring positively and negatively only once. If $v \notin \text{var}(C)$, then we can apply the induction hypothesis to F' obtained from F by (singular) DP-resolution on v . If on the other hand there is no such v , then F is renamable to a Horn clause-set, so, since F is unsatisfiable, F must contain a unit-clause $\{x\} \in F$, and we can apply the induction hypothesis to F'' obtained from F by (singular) DP-resolution on $\text{var}(x)$.

Regarding the intersection of the autarky system of balanced autarkies with the autarky systems presented until now, *balanced linear autarkies* seem most important (that is, linear autarkies which are also balanced, constituting a strong autarky system).

Example 14.11.6. The clause-set $F = \{\{v_1, \bar{v}_2, \bar{v}_3\}, \{\bar{v}_1, v_2, \bar{v}_3\}, \{\bar{v}_1, \bar{v}_2, v_3\}\}$ from Example 14.11.5 is balanced satisfiable (for example by setting all variables to 1 or setting all variables to 0), while F is balanced linearly lean (has no non-trivial balanced lean autarky).

Balanced linear autarkies for a clause-set F can be found easier than general linear autarkies, by just solving a system of linear equations (not linear inequalities), namely they correspond to the solution of $M(F) \cdot x = 0$ (recall Subsection 14.11.3). As already alluded to in Subsection 14.4.5, they have been implicitly used in [Sey74]: While matching autarkies are used to show that minimally unsatisfiable clause-sets have a deficiency at least 1, balanced linear autarkies are used to show that minimally non-2-colourable hypergraphs have deficiency at least 0 (where we define the deficiency of a hypergraph (V, E) as $|E| - |V|$, the difference of the number of hyperedges and the number of vertices). Slightly more general, any clause-set F which is balanced linearly lean has $d(F) \geq 0$, which follows directly from the matrix characterisation of balanced linear autarkies; actually for such F we have the stronger property $d^*(F) = d(F)$ (Lemma 7.2 in [Kul06]). For more on the relation of balanced autarkies to combinatorics see Subsections 14.12.1, 14.12.2.

14.11.5. Generalising minimally unsatisfiable clause-sets

In [Kul07b] the notion of **minimally \mathcal{A} -unsatisfiable** clause-sets F has been introduced for arbitrary (normal) autarky systems \mathcal{A} , which are clause-sets F which are \mathcal{A} -unsatisfiable but where every strict sub-clause-set is \mathcal{A} -satisfiable. The motivation is as follows: The hypergraph 2-colouring problem for a hypergraph $G = (V, E)$ (where V is a finite set of vertices, and E is a set of subsets of V) can be naturally expressed as the satisfiability problem for the clause-set $F_2(G)$ (as already mentioned in Subsections 14.4.5 14.11.4), using the elements of V as variables, while the clauses are the hyperedges $H \in E$ itself (which become now “positive clauses”) together with the complements \bar{H} (which become “negative clauses”). The notion of *minimally non-2-colourable hypergraphs* (or *critically 3-colourable hypergraphs*) has been studied intensively in the literature (see for example Chapter 15 in [JT95]). It is easy to see that G is minimally non-2-colourable if and only if $F_2(G)$ is minimally unsatisfiable. Analogously to clause-sets, for hypergraphs G we can consider the *deficiency* $d(G) := |E(G)| - |V(G)|$. It has been shown in [Sey74] that $d(G) \geq 0$ holds for minimally non-2-colourable hypergraphs. The situation is very similar to $d(F) \geq 1$ for minimally unsatisfiable clause-sets, and indeed, while the proof of this property for clause-sets relies on matching autarkies, for hypergraphs we use balanced linear autarkies instead (thus replacing matching theory by linear algebra; see [Kul06] for some initial investigations on this subject). Deciding the class MU(1) in polynomial time has been discussed in Subsection 14.2.2, while the analogous problem for

hypergraphs, deciding whether a *square hypergraph* (of deficiency 0) is minimally 2-colourable, is actually a much more difficult problem, which was finally resolved in [RST99, McC04] (see Subsection 14.12.2). Now satisfiability of the elements of $\text{MU}(1)$ after removal of a clause is of a very special kind, namely they are *matching satisfiable*, and analogously satisfiability of square minimally non-2-colourable hypergraphs after removal of a hyperedge is covered by *balanced linear satisfiability* (regarding the translation $F_2(G)$). So we have several natural and important cases of a restricted kind of minimal unsatisfiability, namely where satisfiability after removal of a clause yields an \mathcal{A} -satisfiable clause-set for some (feasible) autarky system \mathcal{A} . And instead of requiring that the whole clause-set must be unsatisfiable, we only need it to be \mathcal{A} -unsatisfiable; thus minimal \mathcal{A} -unsatisfiability at the same time strengthens and weakens the notion of minimal unsatisfiability.

Regarding the full autarky system Auk , minimal Auk -unsatisfiability is just minimal unsatisfiability. And every minimally unsatisfiable clause-set is minimally \mathcal{A} -unsatisfiable for every autarky system \mathcal{A} . The minimally matching unsatisfiable (i.e., minimally MAuk -unsatisfiable) clause-sets F are exactly the matching lean clause-sets of deficiency 1, which includes $\text{MU}(1)$, while all other such F are satisfiable.

We have already seen another generalisation of (ordinary) minimal unsatisfiability, namely the notion of \mathcal{A} -lean clause-sets. Now minimal unsatisfiable clause-sets F are not just lean, but they are “minimally lean” in the sense that every strict sub-clause-set except of \top is not lean, so one could introduce the notion of “minimally \mathcal{A} -lean clause-sets” — however the reader can easily convince himself that a clause-set F is “minimally \mathcal{A} -lean” iff F is minimally \mathcal{A} -unsatisfiable. So to arrive at a new notion, we have to be more careful, and instead of demanding that every strict sub-clause-set of F except of \top is not \mathcal{A} -lean, we require that after removal of any *single* clause we obtain a non- \mathcal{A} -lean clause-set. We have arrived at the notion of a **barely \mathcal{A} -lean** clause-set, which generalises the notion of a “barely L -matrix” in [BS95] (see Subsection 14.12.1 below).

Generalising the notion of an “ L -indecomposable matrix” in [BS95] (again, compare Subsection 14.12.1), in [Kul07b] the notion of an **\mathcal{A} -indecomposable clause-set** has been introduced. This notion applies to \mathcal{A} -lean clause-sets, where an \mathcal{A} -lean clause-set F by definition is \mathcal{A} -decomposable if the clause-variable matrix $M(F)$ can be written as $\begin{pmatrix} A & 0 \\ * & B \end{pmatrix}$ for non-empty matrices A, B (having at least one row and one column) corresponding to clause-sets F_1, F_2 which are \mathcal{A} -lean (then the pair (F_1, F_2) yields an \mathcal{A} -decomposition of F). Note that since we are dealing here with clause-sets, B may not contain repeated rows (as in Definition 4 of [Kul07b]), which is justified when only Theorem 14.11.1 is considered, since if F is barely \mathcal{A} -lean, then no contraction can occur in an \mathcal{A} -decomposition.

Example 14.11.7. Consider two (different) variables a, b and the clause-set $F := F_1 \cup F_2'$, where $F_1 := \{\{a\}, \{\bar{a}\}\}$ and $F_2' := \{\{a, b\}, \{\bar{a}, \bar{b}\}\}$. We consider the full autarky system. Now F is barely lean, but not minimally unsatisfiable, and an autarky-decomposition of F is given by F_1, F_2 for $F_2 := F[\{b\}] = \{\{b\}, \{\bar{b}\}\}$. The clause-set $F' := F_1 \cup F_2''$ for $F_2'' := F_2' \cup \{\{a, \bar{b}\}\}$ has no autarky-decomposition as a clause-set (but see below), and it is lean but not barely lean.

The basic theorem now is ([Kul07b]):

Theorem 14.11.1. *Consider a normal autarky system \mathcal{A} . Then a clause-set F with at least two clauses is minimally \mathcal{A} -unsatisfiable if and only if F is barely \mathcal{A} -lean and \mathcal{A} -indecomposable.*

If the autarky system \mathcal{A} is sensitive to repetition of clauses (as are matching autarkies; recall Example 14.11.3), then multi-clause-sets or even labelled clause-sets (which additionally allow for formal variables and for clause-labels; recall Subsection 14.11.3.2) are needed. For the above definition of \mathcal{A} -(in)decomposability then labelled clause-sets are used instead of clause-sets, and so the submatrices A, B may contain zero columns and repeated rows (but note that if A or B contains a zero column then the corresponding labelled clause-set cannot be \mathcal{A} -lean).

Example 14.11.8. Considering $F' = \{\{a\}, \{\bar{a}\}, \{a, b\}, \{\bar{a}, \bar{b}\}, \{a, \bar{b}\}\}$ from Example 14.11.7 as a multi-clause-set, it has now the autarky-decomposition consisting of $\{a\} + \{\bar{a}\}$ and $\{b\} + 2 \cdot \{\bar{b}\}$. And the clause-set $\{\{a\}, \{\bar{a}\}, \{a, b\}, \{\bar{a}, \bar{b}\}\}$ is matching lean (but not barely matching lean), and as a clause-set it has no matching-autarky-decomposition, while as a multi-clause-set it has the matching autarky decomposition consisting of $\{a\} + \{\bar{a}\}$ and $2 \cdot \{b\}$.

An \mathcal{A} -lean labelled clause-set $F = (V, F_0, F^*)$ has an \mathcal{A} -autarky decomposition if and only if there exists $\emptyset \subset V' \subset V$ such that the clause-label-set $F'_0 := \{C \in F_0 : \text{var}(F^*(C)) \subseteq V'\}$ is not empty and the labelled clause-set $F_1 := (V', F'_0, F^*|_{F'_0})$ is \mathcal{A} -lean (this corresponds to the matrix A in the above definition, while $F_2 = F[V \setminus V']$; note that leanness of F_2 is implied by leanness of F , and that in general zero rows in B can be moved to A).

14.11.6. The basic algorithmic problems

To conclude this section on autarky systems, we collect now the basic algorithmic problems concerning autarkies. First we need notations for the various classes of clause-sets involved; consider a class $\mathbb{C} \subseteq \mathcal{CLS}$ of clause-sets and a normal autarky system \mathcal{A} . By \mathcal{A} -SAT(\mathbb{C}) resp. \mathcal{A} -UNSAT(\mathbb{C}) we denote the \mathcal{A} -satisfiable resp. \mathcal{A} -unsatisfiable clause-sets in \mathbb{C} , by \mathcal{A} -LEAN(\mathbb{C}) the \mathcal{A} -lean clause-sets in \mathbb{C} , by \mathcal{A} -MU(\mathbb{C}) the minimally \mathcal{A} -unsatisfiable clause-sets in \mathbb{C} , and finally by \mathcal{A} -BRLEAN(\mathbb{C}) resp. \mathcal{A} -INDEC(\mathbb{C}) the barely \mathcal{A} -lean clause-sets resp. the \mathcal{A} -indecomposable clause-sets in \mathbb{C} . If \mathcal{A} is not mentioned, then the default is the full autarky system, and if \mathbb{C} is not mentioned, then the default is the class \mathcal{CLS} of all clause-sets. Thus $\text{MU}(k) = \text{MU}(\{F \in \mathcal{CLS} : d(F) = k\})$, while LEAN is the set of all lean clause-sets.

The basic *decision problems* are now as follows, as “promise problems”, that is, where $F \in \mathbb{C}$ is already given:

1. The AUTARKY EXISTENCE PROBLEM for \mathcal{A} and \mathbb{C} is to decide whether $F \in \mathcal{CLS} \setminus \mathcal{A}$ -LEAN(\mathbb{C}) holds, while its negation, the LEANNESS PROBLEM, is to decide $F \in \mathcal{A}$ -LEAN(\mathbb{C}).
2. The SATISFIABILITY PROBLEM for \mathcal{A} and \mathbb{C} is to decide whether $F \in \mathcal{A}$ -SAT(\mathbb{C}) holds, while its negation, the UNSATISFIABILITY PROBLEM, is to decide $F \in \mathcal{A}$ -UNSAT(\mathbb{C}).
3. The MINIMAL UNSATISFIABILITY PROBLEM for \mathcal{A} and \mathbb{C} is to decide whether $F \in \mathcal{A}$ -MU(\mathbb{C}) holds.

4. The BARELY LEANNESS PROBLEM is to decide $F \in \mathcal{A}\text{-BRLEAN}(\mathbb{C})$.
5. The INDECOMPOSABILITY PROBLEM is to decide $F \in \mathcal{A}\text{-INDEC}(\mathbb{C})$.

If we can solve the satisfiability problem “efficiently”, and \mathbb{C} is stable under formation of sub-clause-sets (i.e., if $F \in \mathbb{C}$ and $F' \subseteq F$, then also $F' \in \mathbb{C}$), then by definition we can solve “efficiently” the minimal unsatisfiability problem. If we can solve “efficiently” the autarky existence problem, and \mathbb{C} is stable under formation of sub-clause-sets, then by definition the barely leanness problem can also be solved “efficiently”.

The most basic *functional problems* are as follows (again for input $F \in \mathbb{C}$):

1. The LEAN KERNEL PROBLEM for \mathcal{A} and \mathbb{C} is the problem of computing the lean kernel $N_{\mathcal{A}}(F)$.
2. The NON-TRIVIAL AUTARKY PROBLEM for \mathcal{A} and \mathbb{C} is the autarky existence problem together with the computation of some non-trivial autarky in case it exists.
3. The QUASI-MAXIMAL AUTARKY PROBLEM for \mathcal{A} and \mathbb{C} is the problem of computing some quasi-maximal autarky.

If we can solve the non-trivial autarky problem “efficiently”, and \mathbb{C} is stable under formation of sub-clause-sets, then by iteration we can solve “efficiently” the quasi-maximal autarky problem. Obviously solving the quasi-maximal autarky problem implies solving the non-trivial autarky problem and the lean kernel problem, and solving the lean kernel problem implies solving the satisfiability problem and the autarky existence problem. Call \mathbb{C} *stable* if \mathbb{C} is stable under formation of sub-clause-sets and under crossing out variables (that is, if $F \in \mathbb{C}$, then for any set V of variables we have $V * F \in \mathbb{C}$); stability implies stability under application of partial assignments and under restrictions.¹⁶ Now the proof of Lemma 8.6 in [Kul03] yields an algorithm, which for stable classes allows to derive from an “efficient” solution of the autarky existence problem an “efficient” solution of the lean kernel problem; however, as already mentioned at the end of the introduction to Section 14.11, in order to solve also the non-trivial autarky problem further conditions on \mathcal{A} are needed.

Theorem 14.10.1 yields a method for computing maximal (general) autarkies for classes \mathbb{C} which are stable under restriction and for which we can solve the satisfiability problem and the unsatisfiability problem in a functional sense. It is not known whether Theorem 14.10.1 can be generalised to more general autarky systems. Regarding the full problems (unrestricted autarkies and unrestricted clause-sets), as already remarked in the introduction to Section 14.10, the autarky existence problem is NP-complete and the leanness problem is coNP-complete. Of course, the (full) satisfiability problem is NP-complete, the (full) unsatisfiability problem coNP-complete, and the full minimal unsatisfiability problem is D^P -complete. Finally, (full) barely leanness decision is D^P -complete ([KMT08]), while for the (full) indecomposability problem it is only known that it is in Π_2 (the second level of the polynomial hierarchy).

¹⁶Expressed in terms of clause-variable matrices, stability of \mathbb{C} corresponds to the property of a class of $\{-1, 0, +1\}$ -matrices to be stable under formation of submatrices (i.e., under elimination of rows and columns). So “stability” is (the in our opinion essential) half of the conditions on a *semicentral* class of clause-sets in the sense of [Tru98] (Section 5.2).

Considering the classes $\mathcal{CLS}(k) := \{F \in \mathcal{CLS} : d^*(F) = k\}$ (and still unrestricted autarkies; note that these classes are stable under sub-clause-formation), the satisfiability problem is not only polynomial-time solvable for fixed k (recall Section 14.2), but it is also fixed-parameter tractable in k ([Sze04]), and furthermore also finding a satisfying assignment resp. a tree resolution refutation is fixed-parameter tractable. On the other hand, the lean kernel problem is known to be decidable in polynomial time for fixed k (Theorem 4.2 in [Kul00a]), but it is not known whether this problem is also fixed-parameter tractable. Whether solving the non-trivial autarky problem is decidable in polynomial time is not known (only for $k = 1$ we know how to compute a quasi-maximal autarky, by first computing a quasi-maximal matching autarky, and then using that a matching-lean clause-set of deficiency 1 is minimally unsatisfiable iff it is unsatisfiable). The basic problem with the classes $\mathcal{CLS}(k)$ is that they are not stable under crossing out variables. Nothing is known here about the indecomposability problem.

Further complexity results regarding (the autarky system of) balanced autarkies (and unrestricted clause-sets) one finds in Subsection 14.12.2.

14.12. Connections to combinatorics

We need some good notions for some important properties of clause-sets in relation to standard notions from combinatorics: The *degree of a literal* l in a clause-set F is the number of clauses $C \in F$ with $l \in C$, while the *degree of a variable* v in a clause-set F is the sum of the degrees of the literals v and \bar{v} . Note that these degrees are the degrees of literals and variables in the *bipartite literal-clause graph* resp. the *bipartite variable-clause graph*, while the length of a clause is its degree in any of these two most basic graphs related to clause-sets. Regarding a whole clause-set, one needs to distinguish between *minimal literal-degree* and *maximal literal-degree* and between *minimal variable-degree* and *maximal variable-degree*. A clause-set F is called *literal-regular* if every literal in $\text{var}(F) \cup \text{var}(\bar{F})$ has the same (literal-)degree, while F is called *variable-regular* if every variable in $\text{var}(F)$ has the same (variable-)degree. Finally F is called *uniform* if all clauses of F have the same length.

A well-known field of activity relating SAT and combinatorics concerns the question how low the maximal variable-degree can be pushed in a k -uniform minimally unsatisfiable clause-set. These investigations started with [Tov84], with the most recent results in [HS05, HS06]. A related extremal problem is considered in [SZ08], where instead of the maximal variable-degree the number of edges in the *conflict graph* of a k -uniform minimally unsatisfiable clause-set is minimised, where the conflict graph of a clause-set F has the clauses of F as vertices, with an edge joining two vertices iff they have at least one conflict.¹⁷

Further differentiations are relevant here: Instead of considering arbitrary minimally unsatisfiable clause-sets, on the one hand we can restrict attention to hitting clause-sets (i.e., the conflict graph is a complete graph), and on the other hand we can require variable-regularity or, stronger, literal-regularity. Another

¹⁷Note that we take the notion of a “graph” in the strict sense, namely where an edge is a 2-subset of the vertex set, and thus loops or parallel edges are not possible.

regularity condition is *conflict-regularity*, where the conflict-multigraph (now allowing parallel edges) is required to be regular. First investigations regarding hitting clause-sets are found in [SS00, SST07], continued in [HK08]. Here we cannot go further into this interesting field, but we will only consider relations to combinatorics which are related to the notion of *deficiency*.

Especially for hitting clause-sets (and generalisations) the field concerning biclique partitioning (and the hermitian rank) is of interest, relating the notion of deficiency to eigenvalues of graphs and quadratic forms; for first explorations of the connections to SAT see [Kul04a, GK05] (further extended in [Kul07a, HK08]). Again due to space constraints we cannot go further in this direction, but we will restrict our attention to relations which directly concern autarkies.

14.12.1. Autarkies and qualitative matrix analysis

“Qualitative matrix analysis” (“QMA”; see [BS95] for a textbook) is concerned with matrices over the real numbers, where matrices A, B are considered equivalent if they have the same sign matrices, i.e., $\text{sgn}(A) = \text{sgn}(B)$ (these are $\{-1, 0, +1\}$ -matrices), and where questions from matrix analysis (starting with solvability of systems of linear equations) are considered “modulo” this equivalence relation.

As an example consider the notion of a non-singular (square) matrix A (over \mathbb{R}): A is non-singular iff $\det(A) \neq 0$, while A is *sign-non-singular* (short: an *SNS-matrix*) iff every matrix B with $\text{sgn}(B) = \text{sgn}(A)$ is non-singular (for example, $\begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix}$ is an SNS-matrix, while $\begin{pmatrix} 1 & 2 \\ 1 & 1 \end{pmatrix}$ is not). We remark that from $\det(A) = \det(A^t)$ it follows that A is sign-non-singular iff its transposition A^t is sign-non-singular. In the subsequent Subsection 14.12.2 we will discuss the connections between SNS-matrices and SAT in more detail.

Motivated by [DD92], in [Kul03] the basic translations between SAT and QMA have been outlined; it seems justified to call QMA “SAT in disguise”, but with a rather different point of view. Recall from Subsections 14.11.3, 14.11.4, that linear autarkies resp. balanced linear autarkies of a clause-set F correspond to the solutions of $M(F) \cdot \vec{x} \geq 0$ resp. $M(F) \cdot \vec{x} = 0$. Taking these systems of (in)equalities only “qualitatively”, that is, the entries of $M(F)$ can be changed when only their signs are preserved, we arrive at exactly all autarkies resp. all balanced autarkies for F (restricted to $\text{var}(F)$). This is the basic connection between SAT and QMA; more precisely, the details are as follows.

The class of all real matrices B with $\text{sgn}(B) = \text{sgn}(A)$ is denoted by $\mathfrak{Q}(A)$ (the “qualitative class of A ”). Recall from Subsection 14.11.3 that linear autarkies φ for clause-sets F correspond to solutions of $M(F) \cdot \vec{x} \geq 0$, where $M(F)$ is the clause-variable matrix, via the association $\vec{x} \mapsto \varphi_{\vec{x}}$ for partial assignments with $\text{var}(\varphi_{\vec{x}}) \subseteq \text{var}(F)$ given by $\text{sgn}_{\varphi_{\vec{x}}}(l) = \text{sgn}(\vec{x}_{\text{var}(l)})$ for literals l over F (that is, $l \in \text{var}(F) \cup \overline{\text{var}(F)}$). By the same translation from real vectors to partial assignments, (arbitrary) autarkies of F correspond to the solutions of $A \cdot \vec{x} \geq 0$ for $A \in \mathfrak{Q}(M(F))$ (all together), while balanced autarkies of F correspond to the solutions of $A \cdot \vec{x} = 0$ for $A \in \mathfrak{Q}(M(F))$. Furthermore, satisfying (partial) assignments for F correspond to the solutions of $A \cdot \vec{x} > 0$ for $A \in \mathfrak{Q}(M(F))$.¹⁸

¹⁸Where “ $a > b$ ” for vectors a, b (of the same length) means that $a_i > b_i$ for all indices i .

Now the basic dictionary between SAT and QMA is as follows, where, corresponding to the convention in [BS95], clause-sets are represented by their variable-clause-matrix, the transposition of the clause-variable matrix (often characterisations in QMA are based on the characterisations of the above qualitative systems $A \cdot \vec{x} \geq 0$, $A \cdot \vec{x} = 0$ and $A \cdot \vec{x} > 0$ via Gordan's transposition theorem (and its variations)):

- Balanced lean clause-sets correspond to *L-matrices* (matrices A such that all $B \in \Omega(A)$ have linearly independent rows, that is, from $\vec{y}^t \cdot B = 0$ follows $\vec{y} = 0$).
- Lean clause-sets correspond to *L⁺-matrices* (matrices A such that for all $B \in \Omega(A)$ from $\vec{y}^t \cdot B \geq 0$ follows $\vec{y} = 0$).
- Unsatisfiable clause-sets correspond to *sign-central matrices* (matrices A such that for all $B \in \Omega(A)$ there is $\vec{x} \geq 0$, $\vec{x} \neq 0$ with $B \cdot x = 0$)¹⁹, while minimally unsatisfiable clause-sets correspond to *minimal sign-central matrices* (sign-central matrices, where elimination of any column destroys sign-centrality).
- Minimally unsatisfiable clause-sets of deficiency 1 (i.e., the elements of MU(1)) correspond to *S-matrices* (matrices A with exactly one column more than rows, which are *L-matrices*, stay *L-matrices* after elimination of any column, and where there exists $\vec{x} > 0$ with $A \cdot \vec{x} = 0$). The saturated elements of MU(1) correspond to *maximal S-matrices* (where changing any 0 to ± 1 renders the matrix a non-*S*-matrix).
- Thus SNS-matrices are exactly the square *L-matrices*. And, as already mentioned in Subsection 14.11.4, from every $F \in \text{MU}(1)$ we obtain an SNS-matrix by choosing $C \in F$ and considering the variable-clause matrix (or the clause-variable matrix) of $F \setminus \{C\}$.

For some more details on these correspondences see [Kul03]. The exploration of the connections between SAT and QMA has only been started yet, and interesting collaborations are to be expected in this direction.

14.12.2. Autarkies and the even cycle problem

Consider the following decision problems:

- Is a square hypergraph minimally non-2-colourable? (the “square critical hypergraph decision problem”)
- Is a square matrix an SNS-matrix? (the “SNS-matrix decision problem”)
- By multiplying some entries of a square matrix A by -1 , can one reduce the computation of the permanent of A to the computation of the determinant of A ? (“Polya's permanent problem”)
- Does a directed graph contain a directed cycle of even length? (the “even-cycle decision problem”)

All these problems are closely linked by direct translations (see [McC04] for a systematic account, containing many more equivalent problems²⁰). Whether any

¹⁹the equivalence to unsatisfiability follows directly from Gordan's transposition theorem

²⁰regarding the traditional combinatorial point of view, which unfortunately excludes SAT

(thus all) of these problems is solvable in polynomial time was a long-standing open problem, finally (positively) solved in [RST99, McC04]. Now by putting SAT in the centre, the equivalence of these problems becomes very instructive.

14.12.2.1. The equivalence of the basic problems

We already mentioned hypergraph colouring in Subsections 14.4.5 and 14.11.5, introducing the deficiency $d(G)$ of a hypergraph and the *reduced deficiency* $d_r(F)$ of a complement-invariant clause-set (as $d_r(F) := d(\mathfrak{V}(F))$, where $\mathfrak{V}(F)$ is the variable-hypergraph of F). So every minimally unsatisfiable complement-invariant clause-set F has $d_r(F) \geq 0$, and the square critical hypergraph decision problem is up to a different encoding the same as the problem of deciding whether a complement-invariant clause-set F with $d_r(F) = 0$ is minimally unsatisfiable, but where F is a “PN-clause-set”, that is, every clause is either positive or negative. Now the standard reduction of arbitrary clause-sets F to PN-clause-sets (introducing a new variable and a new binary clause for each “mixed” literal-occurrence), symmetrised by using the same variable for a clause C and its complement \bar{C} , translates complement-invariant clause-sets into complement-invariant PN-clause-sets while maintaining the reduced deficiency, and thus we see that by this simple transformation the square critical hypergraph decision problem is the same problem as to decide whether an (arbitrary) complement-invariant clause-sets of reduced deficiency 0 is minimally unsatisfiable.

By the results of Subsection 14.12.1, the SNS-matrix decision problem is (nearly) the same problem as to decide whether a clause-set F with $d(F) = 0$ is balanced lean²¹, which is the same problem as to decide whether a complement-invariant clause-set F with $d_r(F) = 0$ is lean. So we need to find out for complement-invariant clause-set F with $d_r(F) = 0$, what the precise relationship is between the problem of deciding leanness and deciding minimal unsatisfiability. Since a clause-set F is minimally balanced unsatisfiable iff $F \cup \bar{F}$ is minimally unsatisfiable, this is the same as the relationship between the problem of deciding balanced leanness and deciding minimal balanced unsatisfiability for clause-sets F_0 with $d(F_0) = 0$,

For clause-sets F with deficiency $d(F) = 1$, minimal unsatisfiability is equivalent to (that is, is implied by) being lean. Analogously one could expect for complement-invariant clause-sets F with $d_r(F) = 0$, that leanness implies minimal unsatisfiability, or equivalently, that balanced leanness for F_0 with $d(F_0) = 0$ implies minimal balanced unsatisfiability — however this is not the case, as the trivial examples $F = \{\{a\}, \{\bar{a}\}, \{b\}, \{\bar{b}\}\}$ resp. $F_0 = \{\{a\}, \{b\}\}$ shows (the reason is roughly that $1+1 = 2$ while $0+0 = 0$, and so a balanced lean F_0 with deficiency zero can contain a non-empty balanced lean strict sub-clause-set with deficiency zero). The first key here is Theorem 14.11.1, which yields that a clause-set F_0 is minimally balanced unsatisfiable iff F is barely balanced lean and balanced indecomposable. The second key is that for balanced lean F_0 with $d(F_0) = 0$ the autarky decompositions (F_1, F_2) w.r.t. balanced autarkies are exactly the \mathcal{A}_0 -autarky decompositions (F_1, F_2) with $d(F_1) = d(F_2) = 0$ of F_0 w.r.t. the smallest normal autarky system \mathcal{A}_0 (note that every clause-set is \mathcal{A}_0 -lean; see Theorem

²¹there are trivial differences due to clause-sets not having multiple clause-occurrences

13, Part 3 in [Kul07b]). Now these \mathcal{A}_0 -decompositions are the decompositions associated to the notion of a “partly decomposable matrix” (see Theorem 13, Part 4 in [Kul07b], and Subsection 4.2 in [BR91]), and thus can be found in polynomial time. So we can conclude now: Given on the one hand an algorithm for deciding whether F_0 with $d(F_0) = 0$ is balanced lean, to decide whether F_0 with $d(F_0) = 0$ is minimally balanced unsatisfiable, we first check whether F_0 is balanced lean, and if this is the case then we check whether $M(F_0)$ is not partly decomposable (that is, whether $M(F_0)$ is “fully indecomposable”) — F_0 is minimally balanced unsatisfiable iff both tests are passed. On the other hand, assume now that an algorithm is given for deciding whether F_0 with $d(F_0) = 0$ is minimally balanced unsatisfiable. Here one needs to add the observation that if (F_1, F_2) is an \mathcal{A}_0 -autarky decomposition of F , then F is \mathcal{A} -lean iff both F_1, F_2 are \mathcal{A} -lean (for any normal autarky system \mathcal{A}). So for given F_0 first we check whether F_0 is minimally balanced unsatisfiable — if this is the case then F_0 is balanced lean. Otherwise we search for an \mathcal{A}_0 -decomposition (A_1, A_2) with $d(A_1) = d(A_2) = 0$ — if no such decomposition exists then F_0 is not balanced lean while otherwise F_0 is balanced lean iff both F_1, F_2 are balanced lean (applying the whole procedure recursively).

Considering the determinant computation via the Leibniz expansion, it is quite straightforward to show that a square matrix A is an SNS-matrix iff $\det(A) \neq 0$ and $\text{per}(|A|) = |\det(A)|$ holds, where $|A|$ is obtained from A by taking absolute values entrywise, and $\text{per}(A) = \sum_{\pi \in S_n} \prod_{i=1}^n A_{i,\pi(i)}$. This establishes the basic connection to Polya’s permanent problem. We remark that for a clause-set F with $d(F) = 0$ the number of matching satisfying assignments for F is $\text{per}(|M(F)|)$. Finally, by considering the decomposition of permutations into cycles, it is also straightforward to show that a reflexive digraph G (every vertex has a loop) has no even (directed) cycle iff its adjacency matrix (a square $\{0, 1\}$ -matrix, with all entries on the main diagonal equal to 1 due to reflexivity of G) is an SNS-matrix.²²

14.12.2.2. Generalisations and strengthenings

Let us now recapitulate the poly-time decision results in the light of the general algorithmic problems from Subsection 14.11.6. We introduced already the convention that for a class $\mathcal{C} \subseteq \mathcal{CLS}$ of clause-sets, $\mathcal{C}(k)$ is the class of $F \in \mathcal{C}$ with $d^*(F) = k$ (note that for $F \in \text{MU}$ we have $d^*(F) = d(F)$). In general, for some function $f : \mathcal{C} \rightarrow \mathbb{R}$ we write $\mathcal{C}_{f=k} := \{F \in \mathcal{C} : f(F) = k\}$, and similarly $\mathcal{C}_{f \leq k}$ and so on; so $\text{MU}(k) = \text{MU}_{d=k}$. The class of complement-invariant clause-sets is denoted by $\text{CI} := \{F \in \mathcal{CLS} : F = \overline{F}\}$, while the autarky system of balanced autarkies is denoted by the prefix “B”.

Let us denote by $\text{MUCI} := \text{MU}(\text{CI})$ the class of all complement-invariant minimally unsatisfiable clause-sets F , and so by $\text{MUCI}_{d_r=k}$ for $k \in \mathbb{N}_0$ the subclass of clause-sets of reduced deficiency k is denoted. We know

$$\text{MUCI} = \bigcup_{k \in \mathbb{N}_0} \text{MUCI}_{d_r=k},$$

²²If we have given an arbitrary $\{0, 1\}$ -matrix A (possibly with zeros on the diagonal), and we want to reduce the SNS-decision problem for A to the even-cycle problem, then we use that if $\det(A) \neq 0$ then by row-permutation of A we can transform A into a matrix with 1-constant main diagonal (while this transformation does not alter the SNS-property).

where $\text{MUCI}_{d_r=0}$ is decidable in polynomial time (as discussed in Subsection 14.12.2.1). Furthermore we denote by $\text{BMU} := \text{B-MU}$ the class of minimally balanced unsatisfiable clause-sets F , and thus by $\text{BMU}(k)$ for $k \in \mathbb{N}_0$ the subclass given by $d(F) = k$ is denoted. So

$$\text{BMU} = \bigcup_{k \in \mathbb{N}_0} \text{BMU}(k),$$

where $\text{BMU}(0)$ is decidable in polynomial time (again, as discussed in Subsection 14.12.2.1). Note that BMU can be phrased as the class of “minimally not-all-equal-unsatisfiable clause-sets”. And note that, as with $\text{MU}(k)$, for $F \in \text{BMU}$ we have $d^*(F) = d(F)$. Analogously to the $\text{MU}(k)$ -hierarchy, the following conjecture seems natural:

Conjecture 14.12.1. The MINIMAL UNSATISFIABILITY PROBLEM for classes of complement-invariant clause-sets with bounded reduced deficiency is solvable in polynomial time. That is, for every $k \in \mathbb{N}_0$ the class $\text{MUCI}_{d_r=k}$ is decidable in polynomial time. Equivalently, all $\text{BMU}(k)$ for $k \in \mathbb{N}_0$ are decidable in polynomial time. Another equivalent formulation is that for all fixed $k \in \mathbb{N}_0$ the problem whether a hypergraph with deficiency k is minimally non-2-colourable is decidable in polynomial time.

Similarly we define $\text{LEANCI} := \text{LEAN}(\text{CI})$ as the class of complement-invariant lean clause-sets, while $\text{BLEAN} := \text{B-LEAN}$ denotes the class of balanced lean clause-sets (recall that for $F \in \text{BLEAN}$ we have $d^*(F) = d(F)$). So $\text{LEANCI} = \bigcup_{k \in \mathbb{N}_0} \text{LEANCI}_{d_r=k}$ and $\text{BLEAN} = \bigcup_{k \in \mathbb{N}_0} \text{BLEAN}(k)$. As discussed in Subsection 14.12.2.1, for $k = 0$ in both cases we have poly-time decision.

Conjecture 14.12.2. The AUTARKY EXISTENCE PROBLEM for classes of complement-invariant clause-sets with bounded reduced deficiency is solvable in polynomial time. That is, for every $k \in \mathbb{N}_0$ the class $\text{LEANCI}_{d_r=k}$ is decidable in polynomial time. Equivalently, all $\text{BLEAN}(k)$ for $k \in \mathbb{N}_0$ are decidable in polynomial time. Another equivalent formulation is that for all fixed $k \in \mathbb{N}_0$ the problem whether a matrix over $\{-1, 0, +1\}$ (or, equivalently, over $\{0, 1\}$) with exactly k more columns than rows is an L -matrix.

It needs to be investigated whether the reductions from Subsection 14.12.2.1 can be generalised, so that we obtain analogously the equivalence of Conjecture 14.12.2 to Conjecture 14.12.1. Finally we note that for $\text{MU}(k)$ we actually have fixed-parameter tractability, and so we also conjecture the strengthening:

Conjecture 14.12.3. The decision problems in Conjecture 14.12.1 are not just poly-time decidable for each (fixed) $k \in \mathbb{N}_0$, but they are also fixed-parameter tractable in this parameter, i.e., there exists a time bound $f(\ell, k)$ for decision of MUCI depending on k and the number ℓ of literal occurrences in the input, which is of the form $f(\ell, k) = \alpha(k) \cdot \ell^\beta$ for some function α and some constant β .

The analogous problem to Conjecture 14.12.2, namely to decide $\text{LEAN}(k)$, is not known to be fixed-parameter tractable, and so we refrain from conjecturing that also for the problems in Conjecture 14.12.2 we have fixed-parameter tractability.

14.12.2.3. Solving SAT problems by autarky reduction

While in the previous Subsection 14.12.2.2 we considered the basic decision problems as outlined in Subsection 14.11.6, we now turn to the functional problems as discussed there. Recall that for the classes $\mathcal{CLS}(k)$ we do not just have minimally unsatisfiability decision, but stronger satisfiability decision in polynomial time (even fixed-parameter tractability), and furthermore we can solve the lean-kernel problem in polynomial time (but currently it is not known whether the non-trivial autarky problem, or, here equivalent, the quasi-maximal autarky problem, can be solved in polynomial time). Different from the approaches used for $\mathcal{CLS}(k)$, which were satisfiability- or unsatisfiability-based, here now our approach is autarky-based, fundamentally relying on the following “self-reducibility conjecture”.

Conjecture 14.12.4. Given a square non-SNS matrix A over $\{-1, 0, +1\}$, one can find in polynomial time a “witness” matrix $B \in \mathfrak{Q}(A)$ over \mathbb{Z} (with the same sign pattern as A) which is singular.

There are two avenues for proving Conjecture 14.12.4: “constructivising” the proofs in [RST99, McC04], or refining the method from [KMT08] for finding an autarky when only given a decision procedure for leanness (the problem here is, similar to the classes $\mathcal{CLS}(k)$, that we do not have stability under crossing out variables nor under addition of unit-clauses). A witness according to Conjecture 14.12.4 yields a non-trivial balanced autarky for the clause-set corresponding to A . In order to make use of this, analogously to d^* we need to introduce the *maximal reduced deficiency* $d_r^*(F)$ for a complement-invariant clause-set F , defined as the maximum of $d_r(F')$ for complement-invariant $F' \subseteq F$; by definition we have $d_r^*(F) \geq 0$. Now [Kul07b] shows the following, using $\text{SATCI} := \text{SAT}(\text{CI})$ for the satisfiable complement-invariant clause-sets and $\text{BSAT} := \text{B-SAT}$ for the balanced satisfiable clause-sets (i.e., not-all-equal satisfiable clause-sets).

Theorem 14.12.1. *Given Conjecture 14.12.4, we obtain poly-time decision of $\text{SATCI}_{d_r^*=0}$, that is, SAT decision for complement-invariant clause-sets with maximal reduced deficiency 0. Equivalently, we obtain poly-time decision of $\text{BSAT}(0)$, that is, NAESAT decision in poly-time for clause-sets F with $d^*(F) = 0$.*

Stronger, we can compute a quasi-maximal autarky for complement-invariant clause-sets F with $d_r^(F) = 0$. And, equivalently, we can compute a quasi-maximal balanced autarky for clause-sets F with $d^*(F) = 0$.*

Given the positive solution of Polya’s problem, Theorem 14.12.1 (that is, the possibility of computing a quasi-maximal autarky) actually is equivalent to Conjecture 14.12.4. The natural generalisation of Conjecture 14.12.4 thus is as follows.

Conjecture 14.12.5. The QUASI-MAXIMAL AUTARKY PROBLEM for classes of complement-invariant clause-sets with bounded maximal reduced deficiency is solvable in polynomial time, that is, for every $k \in \mathbb{N}_0$ there exists a poly-time procedure for computing for complement-invariant clause-sets F with $d_r^*(F) = k$ a quasi-maximal autarky. Equivalently, quasi-maximal balanced autarkies can be computed in poly-time for $d^*(F) = k$. Or, equivalently, there exists a polynomial-time procedure for deciding whether a matrix A over $\{-1, 0, +1\}$ with (exactly)

k more columns than rows is an L -matrix, and computing in the negative case a singular matrix $B \in \mathfrak{Q}(A)$ over \mathbb{Z} .

As discussed in general in Subsection 14.11.6, on the one hand Conjecture 14.12.5 implies poly-time decision for all $\text{SATCI}_{d_r^*=k}$, and, equivalently, poly-time decision of all $\text{BSAT}(k)$ (that is, NAESAT decision for clause-sets with maximal deficiency k), and thus implies Conjecture 14.12.1. And on the other hand Conjecture 14.12.5 implies also Conjecture 14.12.2.

Finally we remark that by the process of “PN-separation”, as discussed at the beginning of Subsection 14.12.2.1, every complement-invariant clause-set F can be transformed into a complement-invariant PN-clause-set F' (i.e., a hypergraph 2-colouring problem) such that F' is equivalent to F w.r.t. the properties of being satisfiable, minimally unsatisfiable and lean, and where $d_r(F') = d_r(F)$ and $d_r^*(F') = d_r^*(F)$. Actually it is easier to perform this transformation on one “core half” of F , a clause-set F_0 with $F = F_0 \cup \overline{F_0}$: then the properties being preserved by $F_0 \rightsquigarrow F'_0$ are balanced satisfiability, minimal balanced unsatisfiability and balanced leanness, and we have $d_r(F'_0) = d_r(F_0)$ and $d_r^*(F'_0) = d_r^*(F_0)$. Thus the various problems discussed in this subsection can be phrased as hypergraph-2-colouring problems, and the matrices occurring can be restricted to $\{0, 1\}$ -matrices.

14.13. Generalisations and extensions of autarkies

In Subsection 14.8.4 we have already commented on the (slight) generalisation given by the notion of a “weak autarky” (compared to (ordinary) autarkies). In this final section we will discuss more substantial generalisations, either concerning the autarky notion itself (in Subsections 14.13.1, 14.13.2), or concerning more general frameworks (in Subsections 14.13.3, 14.13.4).

14.13.1. Safe partial assignments

The most general case of “safe assignments” are *satisfiability-preserving partial assignments* φ , i.e., where $\varphi * F$ is satisfiability equivalent to F . This poses very mild restrictions on φ : if F is unsatisfiable, then every φ is satisfiability-preserving, while φ is satisfiability-preserving for satisfiable F iff there exists a satisfying (total) assignment extending φ . If φ is satisfiability-preserving for F , then also every $\varphi' \subseteq \varphi$ is satisfiability-preserving for F , and for satisfiable F the satisfiability-preserving partial assignments are exactly the $\varphi' \subseteq \varphi$ for the total satisfying assignments φ .

A satisfiability-preserving partial assignment amounts to specifying certain additional constraints which do not destroy satisfiability. The analogous notion of a *satisfiability-preserving clause* for a clause-set F is a clause C such that $F \cup \{C\}$ is satisfiability-equivalent to F . Examples for satisfiability-preserving clauses for F are clauses which follow logically from F . If C is satisfiability-preserving for F , then also every super-clause of C is satisfiability-preserving for F . Any C is a satisfiability-preserving clause for any unsatisfiable F , while C is satisfiability-preserving for a satisfiable F if and only if there exists $x \in C$ such that $\langle x \rightarrow 1 \rangle$

is satisfiability-preserving for F . So for a literal x the clause $\{x\}$ is satisfiability-preserving for F iff the partial assignment $\langle x \rightarrow 1 \rangle$ is satisfiability-preserving for F , and satisfiability-preservation of clauses in general weakens these cases, while satisfiability-preservation of partial assignments strengthens them.

Lemma 14.13.1. *Consider a clause-set F and compatible partial assignments φ, ψ such that φ is satisfiability-preserving for $\psi * F$. Then for every literal $x \in C_\varphi^1$ the clause $C_\psi^0 \cup \{x\}$ is satisfiability-preserving for F .*

Proof. If $\psi * F$ is unsatisfiable, then C_ψ^0 follows from F , and thus C_ψ^0 is satisfiability-preserving for F . If $\psi * F$ is satisfiable, then also $(\varphi \circ \psi) * F$ is satisfiable, thus $\varphi \circ \psi = \psi \circ \varphi$ is satisfiability-preserving for F , and so is $\langle x \rightarrow 1 \rangle$. \square

The composition of satisfiability-preserving partial assignments in general is not again satisfiability-preserving. The notion of a *safe partial assignment* provides a restriction having this property, where φ is safe for a clause-set F if for every partial assignment ψ with $\psi * F = \top$ we have $\psi * (\varphi * F) = \top$ as well (i.e., if ψ is a satisfying assignment, then so is $\psi \circ \varphi$). Every weak autarky is safe, for unsatisfiable clause-sets again every partial assignment is safe, and more generally every enforced partial assignment φ (i.e., every satisfying total assignment for F is an extension of φ) is safe. Every safe partial assignment is satisfiability-preserving. The safe partial assignments for F yield a monoid (with $\text{Auk}(F)$ as sub-monoid), as is easily checked (using just the fundamental properties of the operation of \mathcal{PASS} on \mathcal{CLS}).

14.13.2. Look-ahead autarky clauses, and blocked clauses

Recall the notion of a weak k -autarky φ for F from Subsection 14.10.5, defined by the condition that for the set $N(\varphi, F) := (\varphi * F) \setminus F$ of new clauses created by the application of φ to F we have $|N(\varphi, F)| \leq k$. For any partial assignment ψ with $\psi * N = \top$ and $\text{var}(\psi) \subseteq \text{var}(N)$ and for every $x' \in C_\varphi^1$ then by Lemma 14.13.1 the *look-ahead autarky clause* $C_\psi^0 \cup \{x'\}$ is satisfiability-preserving for F . In the special case of $\varphi = \varphi_x$, as discussed in Subsection 14.10.5, the choice $x' = x$ is appropriate. For a look-ahead autarky clause we actually can allow a slightly more general case, just given arbitrary partial assignments ψ, φ with $\text{var}(\varphi) \cap \text{var}(\psi) = \emptyset$ such that φ is a weak autarky for $\psi * F$: In this case ψ must satisfy all clauses of $N(\varphi, F)$ except of those which after application of ψ are already contained in $\psi * F$.

Recall the notion of a *blocked clause* for clause-set F w.r.t. $x \in C$ (introduced in [Kul99b], and further studied in [Kul99c]), that is, for every clause D in F with $\bar{x} \in D$ there is $y_D \in C \setminus \{x\}$ with $\overline{y_D} \in D$. Let $C_0 := C \setminus \{x\}$, and consider the partial assignment $\varphi_{C_0} := \langle y \rightarrow 0 : y \in C_0 \rangle$. Now the blocking-condition is equivalent to the literal x being pure for $\varphi_{C_0} * F$, and thus C is a look-ahead autarky clause for F . So look-ahead autarky clauses generalise blocked clauses, where for the latter it is shown in [Kul99c] that their addition to clause-sets cannot be simulated by (full) resolution (even if we only use C without new variables).

Thus addition of look-ahead autarky clauses by a SAT solver could be an interesting feature; though one would expect that the full power of extended

resolution is only unleashed by allowing blocked clauses with new variables, nevertheless the special case of look-ahead autarky clauses considered above, given by some partial assignment φ_x we naturally have “at hand” in a SAT solver, could become interesting in the future. In the `OKsolver-2002` the special case of a weak 1-autarky has been implemented as an experimental feature: if $C = \{a_1, \dots, a_k\}$ ($k \geq 2$) is the (single) new clause, then for $1 \leq i \leq k$ the k binary clauses $\{\overline{a_i}, x\}$ can be added.

Finally we should remark that addition of look-ahead autarky clauses is a form of “local learning”, that is, the added clauses are valid only at the current (“residual”) node in the backtracking tree, and need to be removed when backtracking.

14.13.3. Beyond clause-sets

Autarkies for generalised clause-sets, using non-boolean variables v and literals “ $v \neq \varepsilon$ ” for (single) values ε in the domain of v , are studied in [Kul07a], and some first applications to hypergraph colouring one finds in [Kul06]. As we have seen in Section 14.12.2, the hypergraph 2-colouring problem is captured by considering complement-invariant clause-sets, which can be reduced to their core halves, that is, to the underlying hypergraphs, by considering balanced autarkies. Now for k -colouring, variables with uniform domain $\mathbb{Z}_k = \{0, \dots, k-1\}$ are considered, and *weak hypergraph-colouring* (no hyperedge is monochromatic) is captured by *weakly-balanced autarkies*, which are autarkies where every touched clause contains at least two different assigned values, while *strong hypergraph-colouring* (every hyperedge contains all colours) is captured by *strongly-balanced autarkies*, autarkies where every touched clause contains all possible k assigned values.

Another route of generalisation is to consider boolean functions (in some representation, for example BDD’s as in [FKS⁺04]) instead of clauses, and, more generally, arbitrary sets of constraints. Just replacing “clause” by “constraint” in the basic condition “every touched clause is satisfied”, and allowing partial assignments for constraints (satisfying the constraint iff all total extensions satisfy it), we obtain a straight-forward generalisation.

Regarding QBF, a powerful framework is here to consider QCNF-problems as satisfiability problems, that is the universal variables are “eliminated” while we seek substitutions for the existential variables by boolean functions in the preceding universal variables such that a tautology is created (that is, every clause becomes a tautology). Now an autarky in this setting is a partial substitution of existential variables by boolean functions (depending on the universal variables on which the existential variable depends) such that the substitution either does not touch a clause or makes it a tautology. The simplest case is to consider only constant functions, that is we search for standard CNF-autarkies of the matrix where all universal variables are crossed out.

14.13.4. An axiomatic theory

A first approach towards an “axiomatic theory”, extending [Kul04b], is given in [Kul01], (in principle) comprising all extensions mentioned in Subsections 14.13.1

and 14.13.3. The basic idea is to consider the operation $*$ of a monoid (\mathcal{I}, \circ, e) of “instantiators” (like partial assignments) on a lower semilattice $(\mathcal{P}, \wedge, \top)$ of “problem instances” (like clause-sets with union). We obtain the induced partial order $P_1 \leq P_2$ for problem instances given by $P_1 \leq P_2 :\Leftrightarrow P_1 \wedge P_2 = P_1$. Now *weak autarkies* for problem instances $P \in \mathcal{P}$ are instantiators $\varphi \in \mathcal{I}$ with $\varphi * P \leq P$, while *autarkies* fulfil the condition $\varphi * P' \leq P'$ for all $P' \leq P$. Given the multitude of notions of autarkies, such a general theory should become indispensable at some time in the future.

14.14. Conclusion

An important structural parameter for satisfiability problems about clause-sets (CNF-formulas) and QBF-formulas is the (maximal) deficiency. Various hierarchies based on the notion of (maximal) deficiency emerged, with some poly-time results established, and most open:

1. For deciding minimally unsatisfiability see Section 14.2, while more generally in Subsection 14.11.6 satisfiability decision (and more) is discussed, and a general framework is given (with still open problems in this context).
2. Deciding the tautology/falsity property for quantified boolean formulas is discussed in Section 14.6.
3. “Not-all-equal satisfiability” (with the normal deficiency), or, equivalently, complement-invariant clause-sets with the reduced deficiency, is considered in Section 14.12.2 (see especially Subsection 14.12.2.3 for open problems and conjectures).

References

- [AL86] R. Aharoni and N. Linial. Minimal non-two-colorable hypergraphs and minimal unsatisfiable formulas. *Journal of Combinatorial Theory, A* 43:196–204, 1986.
- [BG06] A. Biere and C. P. Gomes, editors. *Theory and Applications of Satisfiability Testing - SAT 2006*, volume 4121 of *Lecture Notes in Computer Science*. Springer, 2006. ISBN 3-540-37206-7.
- [BK92] A. Bachem and W. Kern. *Linear Programming Duality: An Introduction to Oriented Matroids*. Universitext. Springer-Verlag, Berlin, 1992. ISBN 3-540-55417-3.
- [Bou89] N. Bourbaki. *Algebra I: Chapters 1-3*. Elements of Mathematics. Springer-Verlag, Berlin, 1989. Second printing, ISBN 3-540-19373-1.
- [BR91] R. A. Brualdi and H. J. Ryser. *Combinatorial Matrix Theory*, volume 39 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 1991. ISBN 0-521-32265-0.
- [Bru03] R. Bruni. Approximating minimal unsatisfiable subformulae by means of adaptive core search. *Discrete Applied Mathematics*, 130:85–100, 2003.
- [Bru05] R. Bruni. On exact selection of minimally unsatisfiable subformulae. *Annals for Mathematics and Artificial Intelligence*, 43:35–50, 2005.

- [BS95] R. A. Brualdi and B. L. Shader. *Matrices of sign-solvable linear systems*, volume 116 of *Cambridge Tracts in Mathematics*. Cambridge University Press, 1995. ISBN 0-521-48296-8.
- [BS01] R. Bruni and A. Sassano. Restoring satisfiability or maintaining unsatisfiability by finding small unsatisfiable subformulae. In H. Kautz and B. Selman, editors, *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001)*, volume 9 of *Electronic Notes in Discrete Mathematics (ENDM)*. Elsevier Science, June 2001.
- [BW05] F. Bacchus and T. Walsh, editors. *Theory and Applications of Satisfiability Testing 2005*, volume 3569 of *Lecture Notes in Computer Science*, Berlin, 2005. Springer. ISBN 3-540-26276-8.
- [DD92] G. Davydov and I. Davydova. Tautologies and positive solvability of linear homogeneous systems. *Annals of Pure and Applied Logic*, 57:27–43, 1992.
- [DDKB98] G. Davydov, I. Davydova, and H. Kleine Büning. An efficient algorithm for the minimal unsatisfiability problem for a subclass of CNF. *Annals of Mathematics and Artificial Intelligence*, 23:229–245, 1998.
- [DE92] M. Dalal and D. W. Etherington. A hierarchy of tractable satisfiability problems. *Information Processing Letters*, 44:173–180, 1992.
- [EIS76] S. Even, A. Itai, and A. Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM Journal Computing*, 5(4):691–703, 1976.
- [FKS02] H. Fleischner, O. Kullmann, and S. Szeider. Polynomial-time recognition of minimal unsatisfiable formulas with fixed clause-variable difference. *Theoretical Computer Science*, 289(1):503–516, November 2002.
- [FKS⁺04] J. Franco, M. Kouril, J. Schlipf, S. Weaver, M. Dransfield, and W. M. Vanfleet. Function-complete lookahead in support of efficient SAT search heuristics. *Journal of Universal Computer Science*, 10(12):1655–1692, 2004.
- [FSW06] M. R. Fellows, S. Szeider, and G. Wrightson. On finding short resolution refutations and small unsatisfiable subsets. *Theoretical Computer Science*, 351(3):351–359, 2006.
- [FV03] J. Franco and A. Van Gelder. A perspective on certain polynomial-time solvable classes of satisfiability. *Discrete Applied Mathematics*, 125:177–214, 2003.
- [GK05] N. Galesi and O. Kullmann. Polynomial time SAT decision, hypergraph transversals and the hermitian rank. In H. H. Hoos and D. G. Mitchell, editors, *Theory and Applications of Satisfiability Testing 2004*, volume 3542 of *Lecture Notes in Computer Science*, pages 89–104, Berlin, 2005. Springer. ISBN 3-540-27829-X.
- [GMP06] É. Grégoire, B. Mazure, and C. Piette. Tracking MUSes and strict inconsistent covers. In *Proceedings of the Formal Methods in Computer Aided Design (FMCAD)*, pages 39–46. IEEE Computer Society, 2006.
- [GMP07a] É. Grégoire, B. Mazure, and C. Piette. Boosting a complete technique to find MSS and MUS thanks to a local search oracle. In M. M. Veloso,

- editor, *Proceedings of IJCAI*, pages 2300–2305, 2007.
- [GMP07b] É. Grégoire, B. Mazure, and C. Piette. MUST: Provide a finer-grained explanation of unsatisfiability. In *Principles and Practice of Constraint Programming (CP)*, pages 317–331, 2007.
- [GT04] E. Giunchiglia and A. Tacchella, editors. *Theory and Applications of Satisfiability Testing 2003*, volume 2919 of *Lecture Notes in Computer Science*, Berlin, 2004. Springer. ISBN 3-540-20851-8.
- [Hir00] E. Hirsch. New worst-case upper bounds for SAT. *Journal of Automated Reasoning*, 24(4):397–420, 2000.
- [HK08] M. Henderson and O. Kullmann. Multiclique partitions of multigraphs, and conflict-regular satisfiability problems with non-boolean variables. In preparation, August 2008.
- [HS05] S. Hoory and S. Szeider. Computing unsatisfiable k -SAT instances with few occurrences per variable. *Theoretical Computer Science*, 337:347–359, 2005.
- [HS06] S. Hoory and S. Szeider. A note on unsatisfiable k -CNF formulas with few occurrences per variable. *SIAM Journal on Discrete Mathematics*, 20(2):523–528, 2006.
- [Hv08] M. J. Heule and H. van Maaren. Parallel SAT solving using bit-level operations. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:99–116, 2008.
- [IM95] K. Iwama and E. Miyano. Intractability of read-once resolution. In *Proceedings Structure in Complexity Theory, Tenth Annual Conference*, pages 29–36. IEEE, 1995.
- [Iwa89] K. Iwama. CNF satisfiability test by counting and polynomial average time. *SIAM Journal on Computing*, 18(2):385–391, April 1989.
- [JT95] T. R. Jensen and B. Toft. *Graph Coloring Problems*. Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, 1995. ISBN 0-471-02865-7.
- [KB99] H. Kleine Büning. An upper bound for minimal resolution refutations. In G. Gottlob, E. Grandjean, and K. Seyr, editors, *Computer Science Logic 12th International Workshop, CSL'98*, volume 1584 of *Lecture Notes in Computer Science*, pages 171–178. Springer, 1999.
- [KB00] H. Kleine Büning. On subclasses of minimal unsatisfiable formulas. *Discrete Applied Mathematics*, 107:83–98, 2000.
- [KBL99] H. Kleine Büning and T. Lettmann. *Propositional Logic: Deduction and Algorithms*. Cambridge University Press, 1999.
- [KBZ02] H. Kleine Büning and X. Zhao. Minimal unsatisfiability: Results and open questions. Technical Report tr-ri-02-230, Series Computer Science, University of Paderborn, University of Paderborn, Department of Mathematics and Computer Science, 2002. wwwcs.uni-paderborn.de/cs/ag-klbue/de/research/MinUnsat/.
- [KL97] O. Kullmann and H. Luckhardt. Deciding propositional tautologies: Algorithms and their complexity. Preprint, 82 pages; the ps-file can be obtained at <http://cs.swan.ac.uk/~csoliver/>, January 1997.
- [KL98] O. Kullmann and H. Luckhardt. Algorithms for SAT/TAUT decision based on various measures. Preprint, 71 pages; the ps-file can be

- obtained from <http://cs.swan.ac.uk/~csoliver/>, December 1998.
- [KLMS06] O. Kullmann, I. Lynce, and J. P. Marques-Silva. Categorisation of clauses in conjunctive normal forms: Minimally unsatisfiable subclause-sets and the lean kernel. In Biere and Gomes [BG06], pages 22–35. ISBN 3-540-37206-7.
- [KMT08] O. Kullmann, V. W. Marek, and M. Truszczyński. Computing autarkies and properties of the autarky monoid. In preparation, October 2008.
- [Kul98] O. Kullmann. Heuristics for SAT algorithms: A systematic study. In *SAT'98*, March 1998. Extended abstract for the Second workshop on the satisfiability problem, May 10 - 14, 1998, Eringerfeld, Germany.
- [Kul99a] O. Kullmann. Investigating a general hierarchy of polynomially decidable classes of CNF's based on short tree-like resolution proofs. Technical Report TR99-041, Electronic Colloquium on Computational Complexity (ECCC), October 1999.
- [Kul99b] O. Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, 223(1-2):1–72, July 1999.
- [Kul99c] O. Kullmann. On a generalization of extended resolution. *Discrete Applied Mathematics*, 96-97(1-3):149–176, 1999.
- [Kul00a] O. Kullmann. An application of matroid theory to the SAT problem. In *Fifteenth Annual IEEE Conference on Computational Complexity (2000)*, pages 116–124. IEEE Computer Society, July 2000.
- [Kul00b] O. Kullmann. Investigations on autark assignments. *Discrete Applied Mathematics*, 107:99–137, 2000.
- [Kul01] O. Kullmann. On the use of autarkies for satisfiability decision. In H. Kautz and B. Selman, editors, *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001)*, volume 9 of *Electronic Notes in Discrete Mathematics (ENDM)*. Elsevier Science, June 2001.
- [Kul02] O. Kullmann. Investigating the behaviour of a SAT solver on random formulas. Technical Report CSR 23-2002, Swansea University, Computer Science Report Series (available from <http://www-compsci.swan.ac.uk/reports/2002.html>), October 2002.
- [Kul03] O. Kullmann. Lean clause-sets: Generalizations of minimally unsatisfiable clause-sets. *Discrete Applied Mathematics*, 130:209–249, 2003.
- [Kul04a] O. Kullmann. The combinatorics of conflicts between clauses. In Giunchiglia and Tacchella [GT04], pages 426–440. ISBN 3-540-20851-8.
- [Kul04b] O. Kullmann. Upper and lower bounds on the complexity of generalised resolution and generalised constraint satisfaction problems. *Annals of Mathematics and Artificial Intelligence*, 40(3-4):303–352, March 2004.
- [Kul06] O. Kullmann. Constraint satisfaction problems in clausal form: Autarkies, minimal unsatisfiability, and applications to hypergraph inequalities. In N. Creignou, P. Kolaitis, and H. Vollmer, editors, *Complexity of Constraints*, number 06401 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer

- Informatik (IBFI), Schloss Dagstuhl, Germany, 2006. <http://drops.dagstuhl.de/opus/volltexte/2006/803>.
- [Kul07a] O. Kullmann. Constraint satisfaction problems in clausal form: Autarkies and minimal unsatisfiability. Technical Report TR 07-055, Electronic Colloquium on Computational Complexity (ECCC), June 2007.
- [Kul07b] O. Kullmann. Polynomial time SAT decision for complementation-invariant clause-sets, and sign-non-singular matrices. In J. P. Marques-Silva and K. A. Sakallah, editors, *Theory and Applications of Satisfiability Testing - SAT 2007*, volume 4501 of *Lecture Notes in Computer Science*, pages 314–327. Springer, 2007. ISBN 978-3-540-72787-3.
- [KX05] H. Kleine Büning and D. Xu. The complexity of homomorphisms and renamings for minimal unsatisfiable formulas. *Annals of Mathematics and Artificial Intelligence*, 43(1-4):113–127, 2005.
- [KZ02a] H. Kleine Büning and X. Zhao. The complexity of read-once resolution. *Annals of Mathematics and Artificial Intelligence*, 36(4):419–435, 2002.
- [KZ02b] H. Kleine Büning and X. Zhao. Polynomial time algorithms for computing a representation for minimal unsatisfiable formulas with fixed deficiency. *Information Processing Letters*, 84(3):147–151, November 2002.
- [KZ03] H. Kleine Büning and X. Zhao. On the structure of some classes of minimal unsatisfiable formulas. *Discrete Applied Mathematics*, 130:185–207, 2003.
- [KZ05] H. Kleine Büning and X. Zhao. Extension and equivalence problems for clause minimal formulae. *Annals of Mathematics and Artificial Intelligence*, 43:295–306, 2005.
- [KZ06] H. Kleine Büning and X. Zhao. Minimal false quantified boolean formulas. In Biere and Gomes [BG06], pages 339–352. ISBN 3-540-37206-7.
- [KZ07a] H. Kleine Büning and X. Zhao. The complexity of some subclasses of minimal unsatisfiable formulas. *Journal on Satisfiability, Boolean Modeling and Computation*, 3:1–17, 2007.
- [KZ07b] H. Kleine Büning and X. Zhao. An extension of deficiency and minimal unsatisfiability of quantified boolean formulas. *Journal on Satisfiability, Boolean Modeling and Computation*, 3:115–123, 2007.
- [KZ08a] H. Kleine Büning and X. Zhao. Computational complexity of quantified Boolean formulas with fixed maximal deficiency. *Theoretical Computer Science*, 407(1-3):448–457, 2008.
- [KZ08b] H. Kleine Büning and X. Zhao, editors. *Theory and Applications of Satisfiability Testing - SAT 2008*, volume 4996 of *Lecture Notes in Computer Science*. Springer, 2008. ISBN 978-3-540-79718-0.
- [Lan02] S. Lang. *Algebra*, volume 211 of *Graduate Texts in Mathematics*. Springer, New York, third edition, 2002. ISBN 0-387-95385-X; QA154.3.L3 2002.
- [Lau06] D. Lau. *Function Algebras on Finite Sets: A Basic Course on Many-*

- Valued Logic and Clone Theory*. Springer Monographs in Mathematics. Springer, 2006. ISBN 3-540-36022-0.
- [LS05] M. H. Liffiton and K. A. Sakallah. On finding all minimally unsatisfiable subformulas. In Bacchus and Walsh [BW05], pages 173–186. ISBN 3-540-26276-8.
- [LS08] M. Liffiton and K. Sakallah. Searching for autarkies to trim unsatisfiable clause sets. In Kleine Büning and Zhao [KZ08b], pages 182–195. ISBN 978-3-540-79718-0.
- [Luc84] H. Luckhardt. Obere Komplexitätsschranken für TAUT-Entscheidungen. In *Frege Conference 1984, Schwerin*, pages 331–337. Akademie-Verlag Berlin, 1984.
- [McC04] W. McCuaig. Pólya’s permanent problem. *The Electronic Journal of Combinatorics*, 11, 2004. #R79, 83 pages.
- [MLA+05] M. N. Mneimneh, I. Lynce, Z. S. Andraus, J. P. Marques-Silva, and K. A. Sakallah. A branch and bound algorithm for extracting smallest minimal unsatisfiable formulas. In Bacchus and Walsh [BW05], pages 467–474. ISBN 3-540-26276-8.
- [Mol05] M. Molloy. Cores in random hypergraphs and boolean formulas. *Random Structures and Algorithms*, 27(1):124–135, 2005.
- [MS85] B. Monien and E. Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics*, 10:287–295, 1985.
- [Oku00] F. Okushi. Parallel cooperative propositional theorem proving. *Annals of Mathematics and Artificial Intelligence*, 26:59–85, 2000.
- [OMA+04] Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov. AMUSE: a minimally-unsatisfiable subformula extractor. In *Proceedings of the 41st annual conference on Design automation (DAC)*, pages 518–523, 2004.
- [Pie91] B. C. Pierce. *Basic Category Theory for Computer Scientists*. Foundations of Computing. The MIT Press, 1991. ISBN 0-262-66071-7.
- [PW88] C. H. Papadimitriou and D. Wolfe. The complexity of facets resolved. *Journal of Computer and System Sciences*, 37:2–13, 1988.
- [RST99] N. Robertson, P. D. Seymour, and R. Thomas. Permanents, Pfaffian orientations, and even directed circuits. *Annals of Mathematics*, 150:929–975, 1999.
- [SD97] M. Shaohan and L. Dongmin. A polynomial-time algorithm for reducing the number of variables in MAX SAT problem. *Science in China (Series E)*, 40(3):301–311, June 1997.
- [Sey74] P. D. Seymour. On the two-colouring of hypergraphs. *The Quarterly Journal of Mathematics (Oxford University Press)*, 25:303–312, 1974.
- [SS00] P. Savický and J. Sgall. DNF tautologies with a limited number of occurrences of every variable. *Theoretical Computer Science*, 238:495–498, 2000.
- [SST07] R. H. Sloan, B. Sörényi, and G. Turán. On k -term DNF with the largest number of prime implicants. *SIAM Journal on Discrete Mathematics*, 21(4):987–998, 2007.
- [SZ08] D. Scheder and P. Zumstein. How many conflicts does it need to be unsatisfiable? In Kleine Büning and Zhao [KZ08b], pages 246–256.

- ISBN 978-3-540-79718-0.
- [Sze01] S. Szeider. NP-completeness of refutability by literal-once resolution. In R. Gore, A. Leitsch, and T. Nipkow, editors, *IJCAR 2001, Proceedings of the International Joint Conference on Automated Reasoning*, volume 2083 of *Lecture Notes in Artificial Intelligence*, pages 168–181. Springer-Verlag, 2001.
 - [Sze03] S. Szeider. Homomorphisms of conjunctive normal forms. *Discrete Applied Mathematics*, 130(2):351–365, 2003.
 - [Sze04] S. Szeider. Minimal unsatisfiable formulas with bounded clause-variable difference are fixed-parameter tractable. *Journal of Computer and System Sciences*, 69(4):656–674, 2004.
 - [Sze05] S. Szeider. Generalizations of matched CNF formulas. *Annals of Mathematics and Artificial Intelligence*, 43(1-4):223–238, 2005.
 - [Tov84] C. A. Tovey. A simplified NP-complete satisfiability problem. *Discrete Applied Mathematics*, 8:85–89, 1984.
 - [Tru98] K. Truemper. *Effective Logic Computation*. A Wiley-Interscience Publication, 1998. ISBN 0-471-23886-4.
 - [Van99] A. Van Gelder. Autarky pruning in propositional model elimination reduces failure redundancy. *Journal of Automated Reasoning*, 23(2):137–193, 1999.
 - [van00] H. van Maaren. A short note on some tractable cases of the satisfiability problem. *Information and Computation*, 158(2):125–130, May 2000.
 - [VO99] A. Van Gelder and F. Okushi. A propositional theorem prover to solve planning and other problems. *Annals of Mathematics and Artificial Intelligence*, 26:87–112, 1999.
 - [vW00] H. van Maaren and J. Warners. Solving satisfiability problems using elliptic approximations — effective branching rules. *Discrete Applied Mathematics*, 107:241–259, 2000.
 - [vW08] H. van Maaren and S. Wieringa. Finding guaranteed MUSes fast. In Kleine Büning and Zhao [KZ08b], pages 291–304. ISBN 978-3-540-79718-0.
 - [ZM04] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable Boolean formula. In Giunchiglia and Tacchella [GT04], pages 239–249. ISBN 3-540-20851-8.

This page intentionally left blank

Chapter 15

Proofs of Unsatisfiability

Marijn J.H. Heule

15.1. Introduction

Satisfiability (SAT) solvers have become powerful tools to solve a broad spectrum of applications. Since the SAT problem is NP-complete, it is easy to validate a satisfiability claim using a certificate in the form of a variable assignment. However, if a solver reports that a formula has *no* solutions, the validation of that claim requires a proof of unsatisfiability. The correctness of SAT solving results is important for applications ranging from hardware verification to proving mathematical theorems. Proof checking is significantly less complex compared to the implementations of state-of-the-art SAT solvers, allowing efficient validation even by formally verified checkers [CFHH⁺17, Lam20]. Proofs of unsatisfiability are useful in several applications, including interpolation [VRN13], extraction of minimal unsatisfiable sets (MUSes) [Nad10], and several areas that rely on SAT solvers, such as theorem proving [AFG⁺11, Web06, WA09, WHH13]. This chapter covers the practical aspects of proofs of unsatisfiability, while Chapter 7 on proof complexity discusses the theoretical aspects in more detail.

Many proof formats have been proposed [ZM03, ES03, Bie08, Van08, WHH14, CFHH⁺17, Lam20, HKB19], and even though they might seem very different, they actually have a lot in common. Essentially all well-known proofs of unsatisfiability are so-called *clausal proofs*: sequences of clauses that are claimed to be redundant with respect to a given formula, where *redundant* means that the addition of a clause preserves the satisfiability status of the formula. The valid addition of the (trivially unsatisfiable) empty clause, usually in the final proof step, shows that the formula is unsatisfiable. In addition, the validity of each step should be efficiently checkable. The main difference between existing proof formats is whether so-called *hints* are included. A hint is information that a checker can use to validate the correctness of a clause-addition step more efficiently. Another difference is whether or not the formula is included in a proof.

The generation of unsatisfiability proofs in practical SAT solving dates back to 2003, when Zhang and Malik integrated logging of resolution steps with hints in the SAT solver **zChaff** [ZM03], while Goldberg and Novikov enhanced the solver **Berkmin** [GN03] with logging of proofs without hints. For clausal proofs based on

the resolution proof system, the hints refer to the antecedents of the resolution rule. Without the antecedents, a checker needs to search for clauses that can be resolved together to justify a clause addition step. Although the antecedents can be determined in polynomial time, this can be expensive in practice. On the other hand, including hints in proofs can make them much larger and render the proof production more complicated and expensive.

In recent years, clausal proofs have also been used to validate results of strong proof systems that go beyond resolution. An example of such a proof system is extended resolution [Tse83], which allows the addition of definitions to a formula. Such definitions consist of multiple clauses. In a clausal proof, these clauses are listed in an arbitrary order and validity is checked per clause. Some strong proof systems allow the addition of clauses for which checking the validity is NP-complete [HKB19] if no hint—usually called *witness* in that context—is provided. As a consequence, witnesses are mandatory whereas hints are not.

Whether or not to include hints in proofs typically depends on the usage of the proofs. For various applications that require resolution proofs, such as interpolation [McM03] or MUS extraction [NRS13], it is common to produce proofs with hints. Proofs without hints are more popular in the context of validating SAT solving results. For example, to check the results of the SAT Competitions or the proofs of mathematical theorems, such as the Erdős Discrepancy Theorem [KL14], the Pythagorean Triples Problem [HKM16], and Keller’s Conjecture [BHMN20]. There are exceptions: proofs without hints have also been studied for interpolation [GV14] and MUS extraction [BHMS14].

Support for proof logging started to become widespread in state-of-the-art solvers such as *Lingeling* [Bie13], *Glucose* [AS13], and *CryptoMiniSAT* [Soo13], in 2013, when the SAT Competition made unsatisfiability proofs mandatory for solvers participating in the unsatisfiability tracks. Practically all sequential solvers that participated in recent SAT Competitions, including the strongest solvers around (such as the three solvers mentioned above) emit proofs *without* hints, and only very few solvers support emitting proofs *with* hints.

The lack of support for proofs with hints is due to the overhead (both in CPU and memory) and the difficulty to represent some techniques used in contemporary SAT solvers in terms of resolution. One such technique is conflict clause minimization [SB09], which requires additional bookkeeping to produce a resolution proof [Van09]. Computing the resolution steps increases the runtime on both satisfiable and unsatisfiable formulas. In contrast, emitting a proof without hints incurs only a negligible overhead and requires just modest modifications for most solvers.¹ Proof validation is only required if the solver claims unsatisfiability.

Support for proof logging has become mainstream in recent years and it is now common practice to produce and validate a proof when using SAT technology to solve hard problems, such as the mathematical problems we mentioned earlier. The development of formally-verified proof checkers further increases the confidence in the correctness of results. The high level of trust in the results is important as SAT solvers are increasingly often used to check the correctness of hardware and software, and such claims should not be based on bugs.

¹A patch to add logging of proofs without hints to *MiniSAT* and *Glucose* is available at <https://github.com/marijnheule/drup-patch>.

15.2. Proof Systems

A proof of unsatisfiability is a sequence of clauses that are *redundant* with respect to a given formula. The most general notion of redundancy is satisfiability preservation, which requires that the addition of a clause to a formula does not affect satisfiability. Under this notion of redundancy, every clause is trivially redundant with respect to every unsatisfiable formula. However, unless $P = NP$, there exists no polynomial-time algorithm that can check the validity of clause-addition steps under this general notion of redundancy. Because of this, several proof systems have been invented that rely on simpler syntactic criteria that guarantee redundancy while still being efficiently checkable. In this section, we discuss several such proof systems that allow the addition of clauses that are learned by practical SAT solvers or generated by typical preprocessing and inprocessing techniques.

15.2.1. Preliminaries and Notation

We consider propositional formulas in *conjunctive normal form* (CNF), which are defined as follows. A *literal* is either a variable x (a *positive literal*) or the negation \bar{x} of a variable x (a *negative literal*). The *complement* \bar{l} of a literal l is defined as $\bar{l} = \bar{x}$ if $l = x$ and $\bar{l} = x$ if $l = \bar{x}$. For a literal l , we denote the variable of l by $\text{var}(l)$. A *clause* is a finite disjunction of the form $(l_1 \vee \cdots \vee l_k)$, where l_1, \dots, l_k are literals. If not stated otherwise, we assume that clauses do not contain *complementary literals*, i.e., a literal and its complement. A *formula* is a finite conjunction $C_1 \wedge \cdots \wedge C_n$, where C_1, \dots, C_n are clauses. Clauses can be viewed as sets of literals and formulas as sets of clauses.

An *assignment* is a function from a set of variables to the truth values 1 (*true*) and 0 (*false*). An assignment is *total* with respect to a formula if it assigns a truth value to all variables occurring in the formula, otherwise it is *partial*. We often denote assignments by the sequences of literals they satisfy. For instance, $x\bar{y}$ denotes the assignment that makes x true and y false. We denote the domain of an assignment α by $\text{var}(\alpha)$. A literal l is *satisfied* by an assignment α if l is positive and $\alpha(\text{var}(l)) = 1$ or if it is negative and $\alpha(\text{var}(l)) = 0$. A literal is *falsified* by an assignment if its complement is satisfied by the assignment. A clause is satisfied by an assignment α if it contains a literal that is satisfied by α . Finally, a formula is satisfied by an assignment α if all its clauses are satisfied by α . A formula is *satisfiable* if there exists an assignment that satisfies it, and *unsatisfiable* otherwise.

We denote the empty clause by \perp and the satisfied clause by \top . Given an assignment α and a clause C , we define $C|_\alpha = \top$ if α satisfies C , otherwise $C|_\alpha$ denotes the result of removing from C all the literals falsified by α . Moreover, for a formula F , we define $F|_\alpha = \{C|_\alpha \mid C \in F \text{ and } C|_\alpha \neq \top\}$. We say that a clause C *blocks* an assignment α if $C = \{x \mid \alpha(x) = 0\} \cup \{\bar{x} \mid \alpha(x) = 1\}$.

A *unit clause* is a clause that contains only one literal. The result of applying the *unit-clause rule* to a formula F is the formula $F|_\alpha$ with α being an assignment that satisfies a unit clause in F . The iterated application of the unit-clause rule to a formula, until no unit clauses are left, is called *unit propagation*. If unit propagation on a formula F yields the empty clause \perp , we say that it derived a

conflict on F . For example, unit propagation derives a conflict on $F = (\bar{x} \vee y) \wedge (\bar{y}) \wedge (x)$ since $F|x = (y) \wedge (\bar{y})$ and $F|xy = \perp$.

Two formulas are *logically equivalent* if they are satisfied by the same total assignments. Two formulas F and F' are *equisatisfiable* if both F and F' are satisfiable or if both F and F' are unsatisfiable. By $F \models F'$, we denote that F implies F' , i.e., every assignment that satisfies F and assigns all variables in $\text{var}(F')$ also satisfies F' . Furthermore, by $F \vdash F'$ we denote that for every clause $(l_1 \vee \dots \vee l_k) \in F'$, unit propagation derives a conflict on $F \wedge (\bar{l}_1) \wedge \dots \wedge (\bar{l}_k)$. If $F \vdash F'$, we say that F *implies F' via unit propagation*. As an example, $(x) \wedge (y) \vdash (x \vee z) \wedge (y)$, since unit propagation derives a conflict on both $(x) \wedge (y) \wedge (\bar{x}) \wedge (\bar{z})$ and $(x) \wedge (y) \wedge (\bar{y})$.

Throughout this chapter we will use the following formula E as an example to explain various concepts:

$$E := (\bar{b} \vee c) \wedge (a \vee c) \wedge (\bar{a} \vee b) \wedge (\bar{a} \vee \bar{b}) \wedge (a \vee \bar{b}) \wedge (b \vee \bar{c})$$

15.2.2. Resolution

The resolution rule [Rob65] states that, given two clauses $C_1 = (x \vee a_1 \vee \dots \vee a_n)$ and $C_2 = (\bar{x} \vee b_1 \vee \dots \vee b_m)$ with a complementary pair of literals (in this case x and \bar{x}), the clause $C = (a_1 \vee \dots \vee a_n \vee b_1 \vee \dots \vee b_m)$ can be inferred by resolving on variable x . We say C is the *resolvent* of C_1 and C_2 and write $C = C_1 \diamond C_2$. C_1 and C_2 are called the *antecedents* of C . C is logically implied by any formula containing C_1 and C_2 . The resolution proof system (RES) infers new clauses using the resolution rule. Chapter 7 covers the resolution proof system in more detail.

A *resolution chain* is a sequence of resolution operations such that the result of each operation is an antecedent of the next operation. Resolution chains are computed from left to right. Notice that the resolution operation is not associative. For example, we have $((a \vee c) \diamond (\bar{a} \vee b)) \diamond (\bar{a} \vee \bar{b}) = (\bar{a} \vee c)$, while $(a \vee c) \diamond ((\bar{a} \vee b) \diamond (\bar{a} \vee \bar{b})) = (c)$.

Let $C := (l_1 \vee l_2 \vee \dots \vee l_k)$ be a clause. We denote by \bar{C} the conjunction $(\bar{l}_1) \wedge (\bar{l}_2) \wedge \dots \wedge (\bar{l}_k)$ of unit clauses. C is called a *reverse unit propagation* (RUP) clause with respect to F , if F implies C via unit propagation [Van08]. Remember that F implies C via unit propagation, denoted by $F \vdash C$, if unit propagation derives a conflict on $F \wedge \bar{C}$. The prototypical RUP clauses are the learned clauses in conflict-driven-clause-learning (CDCL) solvers, which are the most common solvers (see Chapter 4 and Sect. 15.3). The conventional procedure for showing that these learned clauses are implied by the formula applies unit propagations in reverse of the order in which they were derived by the CDCL solver, which explains the name *reverse* unit propagation.

Clearly, if C is a RUP clause with respect to a formula F , then F implies C . This is an easy consequence of the fact that F implies C via unit propagation. In fact, it can be shown that one can construct a resolution chain for C using at most $|\text{var}(F)|$ resolutions. For example, $E \wedge (\bar{c}) \vdash (\bar{b}) \vdash (a) \vdash \perp$ uses the clauses $(\bar{b} \vee c)$, $(a \vee c)$, and $(\bar{a} \vee b)$. We can convert this into a resolution chain $(c) := (\bar{a} \vee b) \diamond (a \vee c) \diamond (\bar{b} \vee c)$.

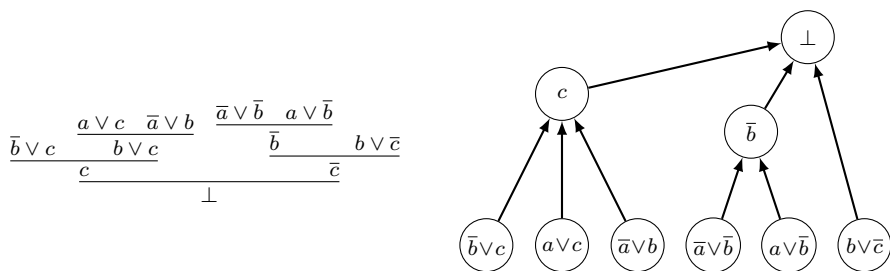


Figure 15.1. A resolution derivation (left) and a resolution graph (right) for the example CNF formula E .

The notion of RUP clauses gives rise to the RUP *proof system*, which consists of a single rule that allows the addition of RUP clauses. More specifically, a RUP proof of a clause C_m from a formula F is a sequence C_1, \dots, C_m of clauses such that $F \wedge C_1 \wedge \dots \wedge C_{i-1} \vdash C_i$ for each $i \in 1, \dots, m$.

15.2.3. Extended resolution and its Generalizations

Extended resolution is a simple but powerful generalization of the ordinary resolution proof system, obtained by adding the so-called *extension rule* [Tse83]. Given a CNF formula F , the extension rule allows one to iteratively add definitions of the form $x \leftrightarrow a \wedge b$ by adding the *definition clauses* $(\bar{x} \vee a) \wedge (\bar{x} \vee b) \wedge (x \vee \bar{a} \vee \bar{b})$ to F , where x is a fresh variable (i.e., a variable that does not occur in the formula or in the previous part of the proof), and a and b are existing literals in the current formula or proof. The proof system of extended resolution (ER) [Tse83] consists of two rules: the resolution rule and the extension rule.

It can be easily seen that the extension rule is sound in the sense that it preserves satisfiability: Every satisfying assignment of a formula *before* addition of the definition clauses can be extended to a satisfying assignment of the formula *after* the addition of the definition clauses by setting the truth value of the fresh variable x to the truth value of $a \wedge b$.

Generalizations of extended resolution, which we discuss later, have a similar property: A satisfying assignment of the formula before addition of a redundant clause can be transformed into a satisfying assignment of the formula after addition of the clause. However, in these more general proof systems it is often not enough to simply extend the existing assignment—sometimes, the truth values of some literals need to be made true. This set of literals is often required as a *witness* (as mentioned earlier) to allow the validation of proofs in polynomial time.

The extended resolution proof system can polynomially simulate extended Frege systems [CR79], which is considered to be one of the most powerful proof systems. Extended resolution proofs can be exponentially smaller compared to resolution proofs. For example, Cook constructed short (polynomial-length) ER proofs for the pigeon hole formulas [Coo76], while Haken proved that resolution proofs of the pigeonhole principle must at least be exponential in size of the

formula [Hak85]. However, searching for ER proofs is challenging as it is difficult to come up with effective applications of the extension rule.

This difficulty of finding useful applications of the extension rule is also a reason why several generalizations of extended resolution have been proposed, in part to make it easier to find short proofs without the need to introduce new variables. The first of these generalizations is the *blocked clause (BC) proof system* [Kul99]. The concept of blocked clauses can be generalized to conditional autarky clauses [HKS17], which give rise to two other proof systems.

15.2.3.1. Blocked Clauses

Given a CNF formula F , a clause C , and a literal $l \in C$, the literal l *blocks* C with respect to F if (i) for each clause $D \in F$ with $\bar{l} \in D$, $C \diamond_l D$ is a tautology, or (ii) $\bar{l} \in C$, i.e., C is itself a tautology. Given a CNF formula F , a clause C is *blocked* with respect to F if there is a literal that blocks C with respect to F . Addition and removal of blocked clauses preserves satisfiability of formulas [Kul99].

Example 15.1. Recall the example formula E . Clause $(\bar{b} \vee c)$ is blocked on c with respect to E , because applying the resolution rule on the only clause containing \bar{c} results in a tautology, i.e., $(\bar{b} \vee c) \diamond (b \vee \bar{c}) = (\bar{b} \vee b)$. Since we know that E is unsatisfiable, $E \setminus \{(\bar{b} \vee c)\}$ must be unsatisfiable.

To see that blocked clause addition is a generalization of extended resolution, consider a formula without variable x , but that contains variables a and b . The three definition clauses from the extension rule, i.e., $(\bar{x} \vee a)$, $(\bar{x} \vee b)$, and $(x \vee \bar{a} \vee \bar{b})$ are all blocked on x or \bar{x} , regardless of the order in which they are added. This is an easy consequence of the fact that x is fresh and that the only resolvents upon x or \bar{x} are obtained by resolving the definition clauses with each other. Hence, blocked clause addition can add these three clauses while preserving satisfiability.

In contrast to extended resolution, blocked clause addition can extend the formula with clauses that are not logically implied by the formula *and* that do not contain a fresh variable. For example, consider the formula $F := (a \vee b)$. The clause $(\bar{a} \vee \bar{b})$ is blocked on \bar{a} (and \bar{b}) with respect to F and can thus be added using blocked clause addition.

15.2.3.2. Conditional Autarkies

An *autarky* is a partial assignment that satisfies all clauses it touches, i.e., all clauses for which the assignment assigns a truth value to at least one literal. Simple examples of autarkies are pure literals and satisfying assignments of a whole formula. Chapter 14 discusses autarkies in detail. A *conditional autarky* [HKS17] is a partial assignment that consists of a so-called *conditional part* and an *autarky part*, where the autarky part is an autarky for the formula restricted under the conditional part.

Example 15.2. Consider the formula $F = (x \vee y) \wedge (\bar{x} \vee z) \wedge (\bar{y} \vee \bar{z})$. The assignment $\omega = \bar{y}z$ is a conditional autarky with conditional part $\omega_{\text{con}} = \bar{y}$: By applying ω_{con} to F , we obtain the formula $F|_{\bar{y}} = (x) \wedge (\bar{x} \vee z)$. The only clause of $F|_{\bar{y}}$ that is touched by ω is the clause $(\bar{x} \vee z)$, which is satisfied by ω . The literal z is a conditional pure literal with respect to ω_{con} .

Conditional autarkies give rise to redundant clauses [KHB19]: If the assignment $c_1 \dots c_m a_1 \dots a_n$ is a conditional autarky (with conditional part $c_1 \dots c_m$) for a formula F , then F and $F \cup \{(c_1 \vee \dots \vee c_m \vee a_i) \mid 1 \leq i \leq n\}$ are equisatisfiable. We refer to the redundant clauses that can be added using this reasoning as conditional autarky clauses. It turns out that blocked clauses are conditional autarky clauses [HKSB17] with the blocking literal being the autarky part, while the other literals form the conditional part of the conditional autarky.

By allowing the autarky part to consist of multiple literals, the notion of blocked clauses can be generalized to *set-blocked clauses* [KSTB16] and *globally-blocked clauses* [KHB19]. Set-blocked clauses are conditional autarky clauses for which all literals in the autarky part are included in the clause. Globally-blocked clauses are conditional autarky clauses for which at least one literal in the autarky part is present in the clause. An example of these clauses is described below.

Example 15.3. Consider again the formula $F = (x \vee y) \wedge (\bar{x} \vee z) \wedge (\bar{y} \vee \bar{z})$. The clause $(x \vee \bar{y} \vee z)$ is a set-blocked clause with respect to F with witness $\omega = x \bar{y} z$. The clause (x) is a globally-blocked clause with respect to F with witness $\omega = x \bar{y} z$. Notice that ω is even an autarky for F .

Deciding if a clause is set-blocked or globally-blocked with respect to a formula is an NP-complete problem [KSTB16]. To efficiently validate the set-blockedness or globally-blockedness property, a witnessing assignment (or witness in short) is required. The witness is the autarky part of corresponding conditional autarky.

The set-blocked clause (SBC) proof system combines the resolution rule with the addition of set-blocked clauses, while the globally-blocked clause (GBC) proof system combines the resolution rule with the addition of globally-blocked clauses.

15.2.4. Strong Proof Systems

In the last decade, several strong proof systems have been proposed that combine the strengths of RUP (easy to emit and compact) with the generalizations of ER (expressive). These proof systems allow for short proofs for hard problems and can compactly express all techniques used in top-tier SAT solvers. The first proof system in this direction is RAT [JHB12]. Two recent generalizations are PR [HKB19] and SR [BT19].

15.2.4.1. Resolution Asymmetric Tautologies

resolution asymmetric tautologies (or RAT clauses) [JHB12] are a generalization of both RUP clauses and blocked clauses. Remember that a blocked clause is a clause that contains a literal such that all resolvents upon this literal are tautologies. If we do not require the resolvents to be tautologies, but instead allow them to be RUP clauses (remember that every tautology is a RUP clause), we obtain the notion of a RAT clause: A clause C is a RAT on a literal l (called the witness or pivot literal) with respect to a formula F if for all $D \in F$ with $\bar{l} \in D$, it holds that $F \vdash C \diamond D$.

It can be shown that the addition and removal of RAT clauses does not affect the satisfiability (or unsatisfiability) of a formula [JHB12]. More specifically, given a formula F and a clause C that is a RAT on $l \in C$ with respect to F , let α be an

assignment that satisfies F and falsifies C . The assignment ω , which is a copy of α with the exception that $\omega(l) = 1$, satisfies $F \wedge C$. While this is not supposed to be obvious, the proof of this observation is quite simple. The observation can be used to reconstruct a satisfying assignment for the original formula in case it is satisfiable. Details about solution reconstruction are described in Section 9.5 in Chapter 9.

We have already discussed why RAT clauses are a generalization of blocked clauses in the sense that every blocked clause is also a RAT clause. It can also be shown that RAT clauses generalize RUP clauses, based on the observation that whenever a clause C is a RUP clause with respect to a formula F , then C is a RAT on each of its literals. Note that this assumes that the considered clause is non-empty. As the empty clause does not contain any literals that could ensure the RAT property, it is often still considered a RAT clause “by definition”.

The RAT proof system consists of a single clause addition rule: the addition of RAT clauses. This proof system extends the RUP proof system by allowing to compactly express techniques that go beyond resolution, including bounded variable addition [MHB12] and symmetry breaking [HHJW15].

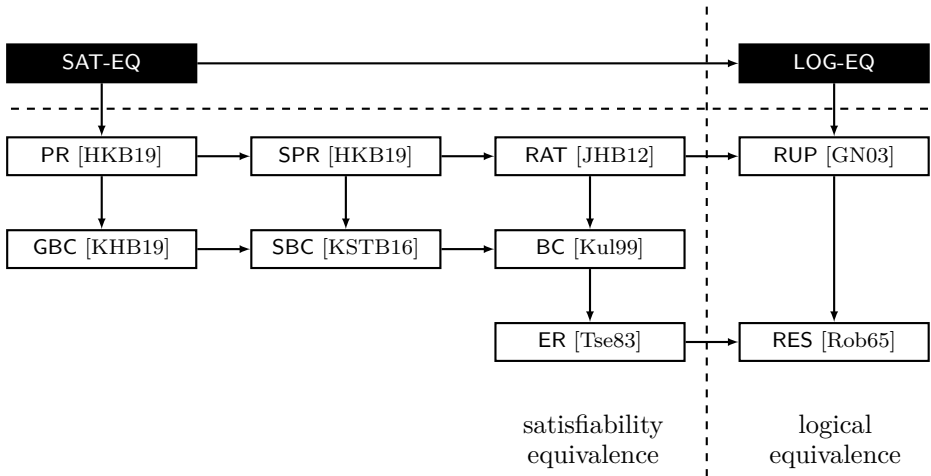


Figure 15.2. Relations between proofs systems for propositional logic. An arrow pointing from proof system A to proof system B means that A is a generalization of B . The black boxes on the top are not actual proof systems, but refer to abstract systems that would allow any clause-addition step that preserves satisfiability (SAT-EQ) or logical equivalence (LOG-EQ).

15.2.4.2. Propagation Redundancy

Another way to express the RAT property is as follows: Given a clause C and a formula F , let α denote the smallest assignment that falsifies C . Now, C is a RAT clause on literal l with respect to F if and only if $F|_{\alpha} \vdash F|_l$ [HKB19]. This notion can be generalized by using a set of literals instead of the single literal l .

A clause C has the *set-propagation redundancy* (SPR) property with respect to a formula F if and only if $F|_{\alpha} \vdash F|_L$, with α denoting the smallest assignment

that falsifies C , and L denoting a set of literals that all occur in C [HKB19]. The set L is the witness of the SPR property of C . SPR clauses generalize both set-blocked clauses and RUP clauses in a similar way that RAT clauses generalize blocked clauses and RUP clauses. The SPR proof system uses a single rule: the addition of SPR clauses. The SPR proof system is surprisingly powerful even when it is restricted such that no new variables are allowed to be introduced, known as SPR^- . For example, there exist short SPR^- proofs for many formulas that are hard for resolution, including the pigeonhole principle, parity principle, Tseitin tautologies, and clique-coloring tautologies [HKB19, BT19].

Finally, the *propagation redundancy* (PR) property generalizes the redundant clauses discussed in this section. A clause C has the PR property with respect to a formula F if and only if $F|\alpha \vdash F|\omega$, with α denoting the smallest assignment that falsifies C , and ω being an arbitrary witness assignment that satisfies C [HKB19]. The PR proof system allows the addition of PR clauses. Figure 15.2 shows a hierarchy of proof systems with PR being the most general one. There exists a polynomial simulation between all proof systems that go beyond logical equivalence, from ER to PR [KRPH18, HB18]. That does not mean that they are similar in practice. A polynomial blowup from PR to ER could mean an enormous increase in validation costs. Also, it is easier to search for proofs in the more general systems as they admit short proofs *without new variables* for some hard problems [HKB19].

15.2.5. Clause Deletion

All the proof systems considered so far in this section add clauses to a given formula. SAT solvers, however, combine both the addition and the deletion of clauses. Most of the discussed proof systems thus have corresponding systems that extend them by allowing the deletion of clauses. The most popular one of these proof systems is probably DRAT, which allows the addition of RAT clauses as well as the deletion of arbitrary clauses. It is important to support the deletion of clauses in proofs without hints for two reasons.

First, it is practically impossible to efficiently validate a proof without hints if no clauses are removed: Due to the lack of hints, the checker needs to find the antecedents, which becomes increasingly expensive as clauses are added to the formula. Even with clause deletion it is generally more expensive to validate the proof compared to its construction. This topic is further discussed in Section 15.6.2.

Second, clause deletion allows for a compact simulation of advanced techniques in solvers. For example, the reasonably compact transformation of a PR proof into a DRAT proof [HB18] is facilitated by the clause deletion step in DRAT. Several clause addition steps would be invalid if some clauses were not removed first. It is possible to simulate the deletion of clauses with extended resolution and thus its generalizations by making a copy of all the clauses apart from the deleted ones using copies (renaming) of the variables [KRPH18]. However, the number of copied clauses is linear in the size of the formula and thus typically much more expensive compared to removal.

In the context of proofs of unsatisfiability, one only needs to check whether clause addition steps are valid. Clause deletion steps can be performed without

a check, because the validity of the unsatisfiability result only depends on clause addition. However, when checking also the validity of clause deletion steps, i.e., whether clause deletion preserves unsatisfiability, one can construct proofs of satisfiability by showing the equivalence of the input formula with the empty formula. This is not really useful in propositional logic as a satisfying assignment can be used as a certificate for satisfiability. Yet, this approach can be useful for richer logics, such as Quantified Boolean Formulas [HSB14], see Chapter 31.

15.3. Proof Search

The leading paradigm to solve satisfiability problems is the conflict-driven clause learning (CDCL) approach, described in Chapter 4. In short, CDCL adds lemmas, typically referred to as conflict clauses, to a given input formula until either it finds a satisfying assignment or it is able to learn (i.e., deduce) the empty clause (prove unsatisfiability).

CDCL solvers typically use a range of preprocessing techniques, see Chapter 9 for details. Examples of such techniques are bounded variable elimination (also known as Davis-Putnam resolution) [DP60, EB05], blocked clause elimination [JBH12], subsumption, and hyper binary resolution [BW03]. Preprocessing techniques are frequently crucial to solve large formulas efficiently. These techniques can also be used during the solving phase, which is known as *inprocessing* [JHB12]. Most preprocessing techniques can be expressed using a few resolutions, such as bounded variable elimination and hyper binary resolution. Other techniques can be ignored in the context of unsatisfiability proofs, because they weaken the formula, such as blocked clause elimination and subsumption.

Some CDCL solvers, however, use preprocessing techniques that are hard to represent using resolution proofs. Examples of such techniques are bounded variable addition [MHB12], blocked clause addition [JHB12], Gaussian elimination, cardinality resolution [CCT87], and symmetry breaking [ASM06]. These techniques cannot be polynomially simulated using the resolution rule: For example, certain formulas based on expander graphs are hard for resolution [Urq87], meaning that they admit only resolution proofs of exponential size, while Gaussian elimination can solve them efficiently. Similarly, formulas arising from the pigeon hole principle are hard for resolution [Hak85], but they can be solved efficiently using either cardinality resolution or symmetry breaking. Consequently, solvers that can efficiently solve such problems cannot produce resolution proofs that are linear in size of the solving time as all resolution proofs are at least exponential in size.

Techniques such as Gaussian elimination, cardinality resolution, and symmetry breaking, can be simulated polynomially using extended resolution and its generalizations. However, it is not known how to simulate these techniques efficiently or elegantly using extended resolution. One method to translate Gaussian elimination into ER proofs is to convert the Gaussian elimination steps into BDDs and afterwards translate the BDDs to a extended resolution proof [SB06].

An alternative approach to solve satisfiability problems is the look-ahead approach, described in Chapter 5. Look-ahead solvers solve a problem via a binary search-tree. In each node of the search-tree, the best splitting variable is

selected using so-called look-ahead techniques. Although it is possible to extract unsatisfiability proofs from look-ahead solvers, this hardly happens in practice. However, look-ahead SAT solvers can also be used to partition a problem into many subproblems, which is known as the *cube-and-conquer* approach [HKWB11]. The subproblems are solved using CDCL. The proofs produced by the CDCL solvers can be merged into a proof of the original problem. This method was used to produce proofs of the Erdős Discrepancy Theorem [KL14], the Pythagorean Triples Problem [HKM16], and Keller’s Conjecture [BHMN20].

Searching for proofs in the stronger proof systems, such as PR, is more complicated and costly as determining whether a clause is a PR clause with respect to a formula is an NP-complete problem. One approach to computing PR proofs is *satisfaction-driven clause learning* (SDCL) [HKS17], which generalizes CDCL by performing a more aggressive pruning of the search tree via the addition of PR clauses.

15.4. Proof Formats

Proof formats define how the proofs of a given proof system should be represented syntactically so that they can be validated by automated tools. Formats for unsatisfiability proofs come in two flavors: with or without hints. A handful of formats have been designed for resolution proofs [ZM03, ES03, Bie08]. All these proof formats include hints in the form of details that describe how to validate each proof step.

The hints in resolution proofs refer to the antecedents of an application of the resolution rule. These formats differ in several details, such as whether the input formula is stored in the proof, whether resolution chains are allowed, or whether hints in the proofs must be ordered. We will first discuss the two most widely used formats with hints: **TraceCheck** and **LRAT**. The **stc** [Bie08] tool can be used to validate **TraceCheck** files. Various checkers exist for **LRAT** files, including **LRAT-check** and the formally verified tool **ACL2-check** [HHKW17]. The **LRAT** format also supports stronger proof systems, but we will first focus on how to use it for resolution proofs.

For proofs without hints, there is essentially only one format, which lists the redundant clauses using the same syntax as **DIMACS**. Such proofs can be extended with clause deletion information [HHW13a], and with a generalization of extended resolution [HHW13b]. The format that supports both extensions is **DRAT** [WHH14], which corresponds to the **DRAT** proof system; it is backward compatible with the **RUP** proof format, which corresponds to the **RUP** proof system. The **DRAT-trim** [WHH14] tool can efficiently validate proofs without hints in the various proof systems.

15.4.1. Formats with Hints

There exist two popular proof formats that include hints: **TraceCheck** and **LRAT**. For resolution proofs, the syntax of both formats is the same. The **TraceCheck** format is restricted to resolution, while the **LRAT** format also allows proofs in stronger proof systems. An important difference between **TraceCheck** and **LRAT**

is that only the former includes the input clauses as part of the proof. In contrast, checkers for LRAT proofs also require the formula as input. The other difference between these formats is that **TraceCheck** allows an arbitrary order of the lines and the hints, while for LRAT the order of lines and hints is strict. These difference can be explained by historical considerations: **TraceCheck** was proposed in 2006, when it was not possible to efficiently validate proofs without hints and when it was also difficult to compute the right order of hints for some techniques. The LRAT format was proposed more than a decade later and was designed to make the format strict to make it easy to develop formally-verified checkers.

The proof checker `tracecheck` can be used to check whether a trace represents a *piecewise regular input resolution proof*, which is also known as a *trivial proof* [BKS04]. A proof is *regular* if variables are resolved at most once along any path in the directed acyclic graph (DAG) formed by the proof. It is an *input resolution* proof if each resolution step resolves at most one non-input clause. Therefore it is also linear and has a degenerated graph structure of a binary tree, where each internal node has at least one leaf as child. A *trace* is just a compact representation of general resolution proofs. The **TraceCheck** format is more compact than other resolution formats, because it uses resolution chains, and because the resulting resolvent does not need to be stated explicitly. The parts of the proof which are regular input resolution proofs are called chains in the following discussion. The full trace consists of original clauses and the chains.

Note that input clauses in chains can still be arbitrary derived clauses with respect to the overall proof and do not have to be original clauses. We distinguish between original clauses of the CNF, which are usually just called input clauses, and input clauses to the chains. Since a chain can be seen as new proof rule, we call its input clauses *antecedents* and the final resolvent just *resolvent*.

The motivation for using this format is that learned clauses in a CDCL solver can be derived by regular input resolution [BKS04]. A unique feature of **TraceCheck** is that the chains do not have to be sorted, neither between chains (globally) nor between their input clauses (locally). If possible the checker will sort them automatically. This allows a simplified implementation of the trace generation.

A chains is simply represented by the list of its antecedents and the resolvent. Intermediate resolvents can be omitted, which saves quite some space if the proof generator can easily extract chains.

Chains can be used in the context of searched-based CDCL to represent the derivations of learned clauses. Computing the chains can become challenging when more advanced learned clause optimizations are used. Shrinking or minimization of learned clauses [SB09] are examples of these optimizations. The difficult part is to order the antecedents correctly. The solver can leave this task to the trace checker instead of changing the minimization algorithm [Van09].

Moreover, both **TraceCheck** and LRAT facilitate a simple encoding of hyper resolution proofs. A hyper resolution step can be simulated by a chain. General resolution steps can also be encoded in this format easily by a trivial chain consisting of the two antecedents of the general resolution step. Finally, extended resolution proofs can directly be encoded, since variables introduced in extended resolution can be treated in the same way as the original variables. Checkers

for TraceCheck don't validate the clauses introduced by the extension rule, while LRAT checkers validate the definition clauses introduced via the extension rule by treating them as blocked-clause-addition steps.

The syntax of a trace is as follows:

$$\begin{aligned} \langle \text{trace} \rangle &= \{ \langle \text{clause} \rangle \} \\ \langle \text{clause} \rangle &= \langle \text{pos} \rangle \langle \text{literals} \rangle \langle \text{antecedents} \rangle \\ \langle \text{literals} \rangle &= "*" \mid \{ \langle \text{lit} \rangle \} "0" \\ \langle \text{hints} \rangle &= \{ \langle \text{pos} \rangle \} "0" \\ \langle \text{lit} \rangle &= \langle \text{pos} \rangle \mid \langle \text{neg} \rangle \\ \langle \text{pos} \rangle &= "1" \mid "2" \mid \dots \mid \langle \text{maxidx} \rangle \\ \langle \text{neg} \rangle &= "-" \langle \text{pos} \rangle \end{aligned}$$

where ‘|’ means choice, ‘{...}’ is equivalent to the Kleene star operation (that is, a finite number of repetitions including 0) and $\langle \text{maxidx} \rangle = 2^{28} - 1$ (originally).

The interpretation is as follows. Original clauses have an empty list of antecedents and derived clauses have at least one antecedent. A clause definition starts with its index and a zero terminated list of its literals, which are represented by integers. This part is similar to the DIMACS format except that each clause is preceded by a unique positive number, the index of the clause. Another zero terminated list of positive indices of its antecedents is added, denoting the chain that is used to derive this clause as resolvent from the antecedents. In TraceCheck, the order of the clauses and the order of the literals and antecedents of a chain is arbitrary. The list of antecedents of a clause should permit a regular input resolution proof of the clause with exactly the antecedents as input clauses. The LRAT format enforces a strict order which will be described below.

input formula (DIMACS)	proof without hints (DRUP)	proof with hints (TraceCheck)
<pre>p cnf 3 6 -2 3 0 1 3 0 -1 2 0 -1 -2 0 1 -2 0 2 -3 0</pre>	<pre>-2 0 3 0 0</pre>	<pre>1 -2 3 0 0 2 1 3 0 0 3 -1 2 0 0 4 -1 -2 0 0 5 1 -2 0 0 6 2 -3 0 0 7 -2 0 4 5 0 8 3 0 1 2 3 0 9 0 7 6 8 0</pre>

Figure 15.3. An input formula (left) in the classical DIMACS format which is supported by most SAT solvers. A proof without hints for the input formula in DRUP format (middle). In both the DIMACS and DRUP formats, each line ending with a zero represents a clause, and each non-zero element represents a literal. Positive numbers represent positive literals, while negative numbers represent negative literals. For example, $-2 \ 3 \ 0$ represents the clause $(\bar{b} \vee c)$. A TraceCheck file (right) is a resolution graph that includes the formula and proof with hints. Each line begins with a clause identifier (bold), then contains the literals of the original clause or lemma, and ends with a list of clause identifiers (bold) as hints.

Consider, for example, the trace shown in Fig. 15.3 (right), which consists of the six clauses from our example CNF formula E , which we introduced on page 638. The corresponding DIMACS file is shown in Fig. 15.3 (left). This proof in `TraceCheck` can easily be converted into LRAT by simply removing the first few lines, which represent the input formula. This is possible as the order of the clauses in the DIMACS file is the same as the order of the original clauses in the trace (which is not required).

The first derived clause in the trace starts with index **7**. This is the unary clause which consists of the literal -2 (\bar{b}). It is obtained by resolving the original clause **4** against the original clause **5** on variable 1 (a).

A chain for the last derived clause, which is the empty clause \perp , can be obtained by resolving the antecedents **7**, **6**, and **8**: first **7** is resolved with **6** to obtain the intermediate resolvent -3 (\bar{c}), which in turn can be resolved with clause **8** to obtain the empty clause \perp .

The LRAT format enforces a strict ordering of the hints. Checking a proof step starts with the assignment that falsifies all literals in the clause. For example, for line **8** it starts with the assignment that makes variable 3 (c) false. The hints are read from left to right and each hint is required to be a unit clause under the current assignment. That assignment is extended by making the unit literal true. The last hint is an exception: the current assignment needs to falsify it. So when checking line **8**, the first hint points to clause $(\bar{b} \vee c)$, which is unit (\bar{b}) and \bar{b} is added to the assignment. The second hint points to $(a \vee c)$, which again is unit (a) and a is added to the assignment. The final hint points to $(\bar{a} \vee b)$, which is indeed falsified by the current assignment.

As discussed above, the order of the lines (clauses) in `TraceCheck` is irrelevant, in contrast to LRAT. The checker will sort them automatically using the clause identifiers. The same holds for the hints. So, the last two lines of the example trace can be replaced by:

```

9 0 7 8 6 0
8 3 0 1 2 3 0

```

Note that the clauses **7** (\bar{b}) and **8** (c) cannot be resolved with each other, because they do not contain complementary literals. In this case, the checker has to reorder the antecedents as in the original example.

The main motivation for having hints in the proof for each learned clause is to speed up proof validation. While checking a learned clause, unit propagation can focus on the list of specified antecedents. It can further ignore all other clauses, particularly those that were already discarded at the point where the solver learned the clause.

In the `TraceCheck` format, it might be convenient to skip the literal part for derived clauses by specifying a $*$ instead of the literal list. The literals are then collected by the checker from the antecedents. Since resolution is not associative, the checker assumes that the antecedents are correctly sorted when $*$ is used.

```

8 * 1 2 3 0
9 * 7 6 8 0

```


Furthermore, trivial clauses and clauses with multiple occurrences of the same literal cannot be resolved. The list of antecedents is not allowed to contain the same index twice. All antecedents have to be used in the proof for the resolvent.

Beside these local restrictions the proof checker generates a global linear order on the derived clauses making sure that there are no cyclic resolution steps. The roots of the resulting DAG are the target resolvents.

One feature in the LRAT format that is currently not supported in `TraceCheck` is the deletion of clauses. Clause deletion lines in LRAT start with an identifier (typically the same identifier as the previous line), followed by “d” and a list of clause indices to be deleted, and they end with a “0”.

15.4.2. Formats Without Hints

Proof formats without hints are designed to make proof logging easy. Apart from the lack of hints, they also don’t use clause indices. The absence of hints and clause indices results in much smaller proofs, but these proofs are more expensive to validate. To validate proofs without hints in reasonable time, it is common to include deletion information. Adding this information is also easy.

We appeal to the notion that *lemmas* are used to construct a proof of a theorem. Here, lemmas represent the learned clauses and the theorem is the statement that the formula is unsatisfiable. From now on, we will use the term clauses to refer to original clauses, while lemmas will refer to added clauses.

$$\begin{aligned}
 \langle \text{proof} \rangle &= \{ \langle \text{lemma} \rangle \} \\
 \langle \text{lemma} \rangle &= \langle \text{delete} \rangle \{ \langle \text{lit} \rangle \} \text{“0”} \\
 \langle \text{delete} \rangle &= \text{“”} \mid \text{“d”} \\
 \langle \text{lit} \rangle &= \langle \text{pos} \rangle \mid \langle \text{neg} \rangle \\
 \langle \text{pos} \rangle &= \text{“1”} \mid \text{“2”} \mid \dots \mid \langle \text{maxidx} \rangle \\
 \langle \text{neg} \rangle &= \text{“-”} \langle \text{pos} \rangle
 \end{aligned}$$

There exist a few proof formats for proofs without hints; they have a very similar syntax as DIMACS and can all be expressed using the above grammar. The most basic format is DRUP, short for delete reverse unit propagation, which combines RUP (reverse unit propagation) additions [Van08] and clause deletions. A DRUP proof is a sequence of lemmas that are either added or deleted. Each lemma is a list of positive and negative integers (to express positive and negative literals, respectively) that terminates with a zero as in DIMACS. Clause deletion steps are expressed using the prefix `d`. An example is shown in Figure 15.3.

Recall that a clausal proof $P := \{C_1, \dots, C_m\}$ is a valid RUP proof for a formula F if $C_m = \perp$ and for each C_i , it holds that

$$F \wedge C_1 \wedge \dots \wedge C_{i-1} \vdash C_i$$

This means that for each $i \in 1, \dots, m$, unit propagation must derive a conflict on the formula $F \wedge C_1 \wedge \dots \wedge C_{i-1} \wedge \bar{C}_i$. Consider again the example CNF

formula E from page 638. A RUP proof is shown in Figure 15.3. This proof, $P_E := \{(\bar{b}), (c), \perp\}$, is a valid proof for E , because P_E terminates with \perp and

$$\begin{array}{ll} E & \vdash_{\top} (\bar{b}) \\ E \wedge (\bar{b}) & \vdash_{\top} (c) \\ E \wedge (\bar{b}) \wedge (c) & \vdash_{\top} \perp \end{array}$$

15.4.3. Formats with Witnesses

So far we only considered proof formats that validate techniques which can be simulated using resolution. As mentioned earlier, however, some SAT solvers use techniques—such as blocked clause addition or symmetry breaking—that cannot be simulated using resolution. To validate these techniques, proof formats need to support stronger proof systems such as extended resolution or one of its generalizations.

In these formats, a witness is provided for each proof step in addition to the clause and the optional hints. For most proof systems, the witness consists of a single literal that needs to be part of the clause. To minimize overhead, the convention is to place the witness as the first literal in the clause. For the clauses added by the extension rule this means to put the literals referring to the new variable first.

Resolution proofs, as the name suggests, can only be used to check techniques that can be expressed using resolution steps. The `TraceCheck` format partially supports extended resolution in the sense that one can add the clauses from the extension rule using an empty list of antecedents. Hence these clauses are considered to be input clauses without actually validating them.

The `DRAT` proof format [WHH14], which is syntactically the same as the `DRUP` format, supports expressing techniques based on extended resolution and its generalizations. The difference between the `DRUP` and `DRAT` format is in the redundancy check that is computed in the checker for proofs in that format. A checker for `DRUP` proofs validates whether each lemma is a RUP clause, while a checker of `DRAT` proofs checks whether each lemma is a RAT clause [HHW13b].

Example 15.4. Consider the following CNF formula

$$G := (\bar{a}\bar{v}\bar{b}\bar{v}\bar{c}) \wedge (a\vee d) \wedge (a\vee e) \wedge (b\vee d) \wedge (b\vee e) \wedge (c\vee d) \wedge (c\vee e) \wedge (\bar{d}\bar{v}\bar{e})$$

On the left of Fig. 15.4, G is shown in the `DIMACS` format, using the conventional mapping from the alphabet to numbers, where a is mapped to 1, \bar{a} is mapped to -1, b to 2, \bar{b} to -2, and so on. In the middle of Fig. 15.4, a `DRAT` proof for G is shown. The proof uses the earlier-mentioned technique of bounded variable addition [MHB12], which cannot be expressed using resolution steps. Bounded variable addition can replace the first six binary clauses by five new binary clauses using a fresh variable f : $(f\vee a)$, $(f\vee b)$, $(f\vee c)$, $(\bar{f}\vee d)$, and $(\bar{f}\vee e)$. These new binary clauses are RAT clauses. Fig. 15.4 shows how to express bounded variable addition in the `DRAT` format: First add the new binary clauses using the pivot (witness) literal on the first position, followed by deleting the old ones. After the replacement, the proof is short $\{(f), \perp\}$.

Validating proofs based on extended resolution or one of its generalizations requires reasoning about absence. For example, the addition of definition clauses via the extension rule is only valid when the new variable is indeed new and does not occur in the formula. We need to check all clauses that are reduced but not satisfied by the witness. In most cases, the witness is a single literal, so we need to check only the clauses in which the complement of the literal occurs.

Consider the following example of a proof that again uses bounded variable addition. Fig. 15.4 shows the example formula G in DIMACS as well as a DRAT proof and an LRAT proof. The clauses $(f \vee a)$, $(f \vee b)$, and $(f \vee c)$ are trivially redundant with respect to G using witness f , because G does not contain any clause with literal \bar{f} . Making f true would satisfy these three clauses. However, $(\bar{f} \vee d)$ and $(\bar{f} \vee e)$ are not trivially redundant with respect to G using witness \bar{f} after the addition of $(f \vee a)$, $(f \vee b)$, and $(f \vee c)$. For example, the redundancy of $(\bar{f} \vee d)$ depends on the presence of $(a \vee d)$, $(b \vee d)$, and $(c \vee d)$ as the RAT check would fail without these clauses.

Let's recall the definition of RAT clauses from Section 15.2.4.1: A clause C is a RAT on l with respect to a formula F if for all $D \in F$ with $\bar{l} \in D$, it holds that $F \vdash C \diamond D$. In the proof, $(\bar{f} \vee d)$ is claimed to be RAT on \bar{f} . Let G' denote the formula when checking $(\bar{f} \vee d)$: $G' := G \wedge (f \vee a) \wedge (f \vee b) \wedge (f \vee c)$. We can confirm the RAT claim by applying the following three checks:

$$\begin{aligned} G' \vdash (a \vee d) & \quad \text{using } D = (f \vee a) \\ G' \vdash (b \vee d) & \quad \text{using } D = (f \vee b) \\ G' \vdash (c \vee d) & \quad \text{using } D = (f \vee c) \end{aligned}$$

The first check succeeds because $(a \vee d) \in G'$ is reduced to the empty clause, whereas the second and third checks succeed, because $(b \vee d) \in G'$ and $(c \vee d) \in G'$ are reduced to the empty clause. In the LRAT proof format this is expressed as follows: For each clause that is reduced but not satisfied by the witness (the D clauses), in this case $(f \vee a)$, $(f \vee b)$, and $(f \vee c)$, the negated clause index is listed: -9, -10, and -11, respectively. After the negated clause index, the clause indices of the unit clauses are listed and finally the clause that is reduced to the empty clause. For $(f \vee a)$, clause $(a \vee d)$ is reduced to the empty clause, which has clause index 2.

15.4.4. Binary Formats and Proof Compression

It is common practice to store proofs on disk, and we discussed various formats for this purpose. However, in many applications where proofs have to be further processed and are used subsequently or even iteratively, disk I/O is considered a substantial overhead. In this subsection, we discuss light-weight options to compress proofs [HB16]. The variable-byte encoding discussed below is supported by most SAT solvers.

15.4.4.1. Byte Encoding

The ASCII encoding of a proof line without hints in Figure 15.3 is easy to read, but rather verbose. For example, consider the literal -123456789, which requires

DIMACS formula	DRAT proof	LRAT proof
p cnf 5 8		
-1 -2 -3 0	6 1 0	9 6 1 0 0
1 4 0	6 2 0	10 6 2 0 0
1 5 0	6 3 0	11 6 3 0 0
2 4 0	-6 4 0	12 -6 4 0 -9 2 -10 4 -11 6 0
2 5 0	-6 5 0	13 -6 5 0 -9 3 -10 5 -11 7 0
3 4 0	d 1 4 0	14 6 0 1 9 10 11 0
3 5 0	d 2 4 0	14 d 2 4 6 3 5 7 0
-4 -5 0	d 3 4 0	15 0 8 12 13 14 0
	d 1 5 0	
	d 2 5 0	
	d 3 5 0	
	6 0	
	0	

Figure 15.4. Example formula G in the classical DIMACS format (left). A proof without hints for the input formula in DRAT format (middle). The corresponding proof with hints as a LRAT file is shown on the right. Notice that the formula is not included. Each line begins with a clause identifier (bold), then contains the literals of the lemma, and ends with a list of hints (bold).

11 bytes to express (one for each ASCII character and one for the separating space). This literal can also be represented by a signed integer (4 bytes). If all literals in a proof can be expressed using signed integers, only 4 bytes are required to encode each literal. Such an encoding also facilitates omitting a byte to express the separation of literals. Consequently, one can easily compress an ASCII proof with a factor of roughly 2.5 by using a binary encoding of literals.

In case the length of literals in the ASCII representation differs a lot, it may not be efficient to allocate a fixed number of bytes to express each literal. Alternatively, the *variable-byte encoding* [WMB99] can be applied, which uses the most significant bit of each byte to denote whether a byte is the last byte required to express a given literal. The variable-byte encoding can express the literal 1234 (10011010010 in binary notation) using only two bytes: 11010010 00001001. (in little-endian ordering, e.g., least-significant byte first).

15.4.4.2. Sorting Literals

The order of literals in a clausal proof has no influence on the validity or the size of a proof. However, the order of literals can influence the cost of validating a proof, as it influences unit propagation and in turn determines which clauses will be marked in backwards checking (the default validation algorithm used in checkers of proofs without hints), see Section 15.6.2 for details. The order of literals in the proof produced by the SAT solver is typically not better or worse than any permutation for validation purposes. However, experience shows that this is often not the case for SAT solving: the given order of literals in an encoding results in stronger solver performance compared to any permutation.

Sorting literals before compression has advantages in both light-weight and heavy compression. In light-weight compression, one can use delta encoding: store the difference between two successive literals. Clauses in a proof are typically long (dozens of literals) [HHW13a], resulting in a small difference between two

successive sorted literals. Delta encoding is particularly useful in combination with variable-byte encoding.

In heavy compression, off-the-shelf zip tools could exploit the sorted order of literals. Many clauses in proofs have multiple literals in common. SAT solvers tend to emit literals in a random order. This makes it hard for compression tools to detect overlapping literals between clauses. Sorting literals potentially increases the observability of overlap which in turn could increase the quality of the compression algorithm.

Table 15.1. Eight encodings of an example DRAT proof line. The first two encodings are shown as ASCII text using decimal numbers, while the last six are shown as hexadecimals using the MiniSAT encoding of literals. The prefix *s* denotes sorted, while the prefix *ds* denotes delta encoding after sorted. *4byte* denotes that 4 bytes are used to represent each literal, while *vbyte* denotes that variable-byte encoding is used.

encoding	example (prefix pivot lit ₁ ...lit _{k-1} end)	#bytes
ascii	d 6278 -3425 -42311 9173 22754 0\n	33
sascii	d 6278 -3425 9173 22754 -42311 0\n	33
4byte	64 0c310000 c31a0000 8f4a0100 aa470000 c4b10000 00000000	25
s4byte	64 0c310000 c31a0000 aa470000 c4b10000 8f4a0100 00000000	25
ds4byte	64 0c310000 c31a0000 e82c0000 1a6a0000 cb980000 00000000	25
vbyte	64 8c62c335 8f9505aa 8f01c4e3 0200	15
svbyte	64 8c62c335 aa8f01c4 e3028f95 0500	15
dsvbyte	64 8c62c335 e8599ad4 01cb b102 00	14

15.4.4.3. Literal Encoding

In most SAT solvers, literals are mapped to natural numbers. The default mapping function $map(l)$, introduced in MiniSAT [ES03] and also used in the AIGER format [Bie07], converts signed DIMACS literals into unsigned integer numbers as follows:

$$map(l) = \begin{cases} 2l + 1 & \text{if } l > 0 \\ -2l & \text{otherwise} \end{cases}$$

Table 15.1 shows a proof line in the DRAT format and in several binary encodings. For all non-ASCII encodings, we will use $map(l)$ to represent literals. Notice that the first literal in the example is not sorted, because the proof checker needs to know the witness / pivot literal (which is the first literal in each clause). The remaining literals are sorted based on their $map(l)$ value.

15.5. Proof Production in Practical SAT Solving

Proof logging of unsatisfiability results from practical SAT solvers started in 2003, with both resolution proofs *with* hints [ZM03] and resolution proofs *without* hints [GN03]. Proofs with hints are typically hard to produce and tend to require lots of overhead in memory, which in turn slows down a SAT solver. Emitting proofs without hints is easy and requires hardly any overhead in memory. We will first describe how to produce proofs without hints, followed by how to produce proofs with hints, and finally describe parallel proof production.

15.5.1. Proofs Without Hints

For all proof formats without hints, SAT solvers simply emit proofs directly to disk. Consequently, there is no memory overhead. In contrast to proofs with hints, it is easy to emit proofs. For the most simple format of proofs without hints, RUP, one only needs to extend the proof with all clauses learned by the CDCL solver. This can be implemented by a handful lines of code. Below we discuss how to produce proofs without hints that support deletion information and techniques based on generalizations of extended resolution.

Proofs without hints need deletion information for efficient validation, see Section 15.6.2 for details. Adding deletion information to a proof without hints is straightforward. As soon as a clause is removed from the solver, the proof is extended by the removed clause using a prefix expressing that it is a deletion step. Recall that clauses in proofs without hints have no clause index. Hence all literals in the deleted clause are listed. The checker needs to find the deleted clause in the clause database. This can be done efficiently using a hash function. If a solver removes a literal l from a clause C , then $C \setminus \{l\}$ is added as a lemma followed by deleting C .

Most techniques based on extended resolution or its generalizations can easily be expressed in proofs without hints. Similar to techniques based on resolution, one simply adds the lemmas to the proof for most techniques. The RAT and DRAT formats only require that the witness / pivot literal is the first literal of the lemma in the proof. However, as discussed in Section 15.3, there exist some techniques for which it is not known whether they can be expressed elegantly in the DRAT format. As mentioned already, especially Gaussian elimination, cardinality resolution, and symmetry breaking are hard to express in the current formats. Finally, for proof systems that require witnesses, the witness is printed directly after the clause and before the zero that marks the end of the line.

15.5.2. Proofs With Hints

Early work on proofs with hints [ZM03] mostly showed that proofs can be generated in principle, but ignored dealing with practical aspects such as keeping proofs compact and supporting all techniques. The first proof format that took these aspects into account was `TraceCheck`, which we discussed already and which was first supported by `PicoSAT` [Bie08]. The addition of proof logging was motivated by making testing and debugging more efficient. In combination with file-based delta-debugging [BLB10], proof trace generation allows to reduce discrepancies much more than without proof tracing.

The original idea was to use resolution traces. However, for some time it was unclear how to extract resolution proofs during a-posteriori clause minimization [SB09]. This was the reason for using a trace format instead: clause minimization is obviously compatible with reverse unit propagation, since required clause antecedents can easily be obtained. However, determining the right order of the hints for a resolution proof is hard to generate directly and probably needs a RUP style algorithm anyhow. It was shown how clause minimization can be integrated in `MiniSAT` using a depth-first search for the first-unique-implication-point clause [Van09], which at the same time can produce a resolution proof for

the minimized learned clause. Currently it is unsolved how to further extend this approach to work with on-the-fly subsumption [HS09] as well. The solution is also not as easy to add to existing solvers as tracing added and deleted clauses.

As already discussed above, memory consumption of proofs stored in memory (or disk) can become a bottleneck. One way to reduce space requirements is to delete garbage clauses that are not used anymore. This garbage collection was implemented with saturating reference counters in *PicoSAT* [Bie08]. It has been shown that full reference counting can result in substantial reductions [ANORC08]. In principle, it might also be possible to use a simple mark-and-sweep garbage collector, which should be faster, since it does not need to maintain and update reference counters.

15.5.3. Parallel Proof Production

A framework based on the cube-and-conquer paradigm has been used to deal with hard combinatorial problems that require thousands of hours of computation to solve [HKM16]. In short, the framework consists of five phases: *encode*, *re-encode*, *split*, *solve*, and *validate*. The focus of the encode phase is to make sure that encoding the problem into SAT is valid. The re-encode phase reformulates the problem to reduce the computation costs of the later phases. The split phase partitions the transformed formula into many (typically millions of) subproblems. The subproblems are solved in the solve phase. The validation phase checks whether the proofs emitted in the prior phases are a valid refutation for the original formula. Figure 15.5 shows an illustration of the framework.

In recent years, three long-standing open problem in mathematics have been solved using this framework: the Pythagorean Triples Problem (PTP) [HKM16], Schur number five (S5) [Heu18], and Keller's Conjecture [BHMN20]. The encoding of PTP and S5 is natural and can be achieved with about a dozen lines of code. The encoding of Keller is more involved and also included several symmetry-breaking clauses that arise from symmetries of the problem, but these are not symmetries in the encoding. The re-encoding phase of all three problems consists of adding symmetry-breaking predicates. Additionally, about 30% of the clauses of PTP are redundant and they have been removed using blocked clause elimination.

All three problems used a two-layered splitting approach. For PTN and S5, the re-encoded problem was split into a few million subproblems using the look-ahead solver *march.cu*. This can be done in less than an hour of a single CPU. For Keller, the problem was manually split into many thousands of subproblems. All these subproblems were split into even smaller problems using *march.cu*. This second layer of splitting was performed in parallel. It resulted in billions of subproblem for PTN and S5 and millions of subproblems for Keller. The subproblems were solved using a CDCL solver. Most subproblems are very easy and almost any CDCL solver could be used for this purpose.

The proofs of these problems consists of three parts: The *re-encoding proof*, the *implication proof*, and the *tautology proof*. These proof part shows the satisfiability equivalence of three formulas: 1) the encoded formula F , 2) the re-encoded formula R , and 3) the negation of all of cubes that are part of the first layer

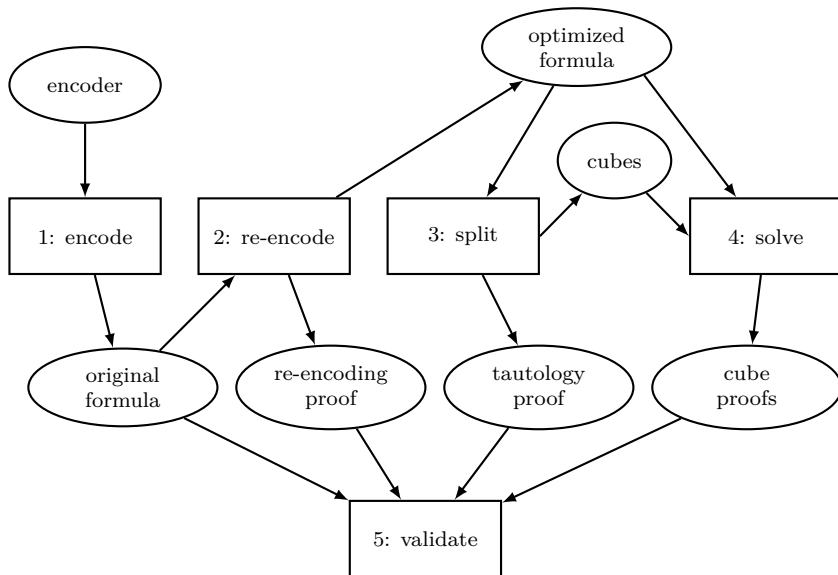


Figure 15.5. Illustration of the framework to solve hard combinatorial problems. The phases are shown in the rectangle boxes, while the input and output files for these phases are shown in oval boxes.

split T . The latter is called T as the cubes need to cover the entire search space, which means that it needs to be a tautology. Together the proof parts show the following relationship:

$$\underbrace{F \models R}_{\text{re-encoding proof}} \models \underbrace{T}_{\text{tautology proof}} \models \perp .$$

implication proof

The re-encoding proof expresses the correctness of the re-encoding phase, which consisted of symmetry breaking and the removal of redundant clauses. The re-encoding cannot be compactly expressed using resolution, so a proof of extended resolution or its generalizations is required. For PTP, S5, and Keller, the re-encoding proof used DRAT.

By far the largest part of the proof (typically over 99%) is the implication proof. It shows that R is unsatisfiable by proving that every clause $C \in T$ is logically implied by R . For each clause $C \in T$ a separate proof piece is produced and these pieces are glued together at the end, which can be done in any arbitrary order. The lack of hints in DRAT proofs make the glueing easy: one can simply concatenate the pieces.

The generation of the final part, the tautology proof, is easy as well as the cubes from the first layer split form a binary tree of assignments by construction. Let m be the number of cubes in our partition, then the tautology proof is a resolution proof consisting of $m-1$ resolution steps.

15.6. Proof Validation and Processing

Although proofs with hints are harder to produce than proofs without hints, they are in principle easy to check and actually needed for some applications, like generating interpolants [McM03]. However, the large size of resolution proofs provides challenges, particularly with respect to memory usage. See [Van12] for a discussion on checking large resolution proofs. The presence of hints makes it easy to implement a checker, especially if the proof format does not allow any ambiguity, such as the LRAT format. This allowed for the implementation of certified (formally verified) checkers, such as ACL2-check [HHKW17].

Proofs without hints are smaller, but validating proofs without hints is more complicated and more costly. Practically all top-tier solvers support the production of proofs without hints, as this is easy to implement, while hardly any of them supports the production of proofs with hints. To obtain the best of both worlds—easy proof production and certified results—there exist tools that turn a proof without hints into a proof with hints. One such tool is DRAT-trim [WHH14], which can turn a DRAT proof into an LRAT proof. DRAT-trim also allows the optimization of proofs, for instance, by removing redundant lemmas. The corresponding tool chain for certifying proofs is shown in Figure 15.6: A SAT solver produces a proof *without* hints, which is turned into an optimized proof *with* hints that is finally validated by a certified checker.

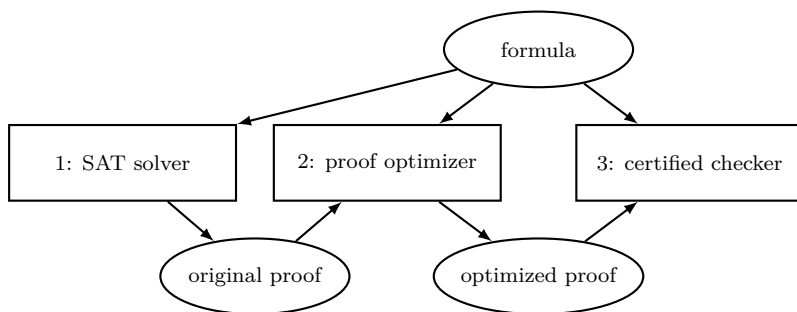


Figure 15.6. Tool chain of certifying proofs.

15.6.1. Proofs With Hints

Resolution proofs can be checked in deterministic logspace [Van08], a very low complexity class. The tool `stc` can efficiently check proofs in the `TraceCheck` format. More details about the format and its use are discussed in Section 15.4.1.

Apart from validation, there exists a vast body of work on compression techniques for resolution proofs [Amj07, BIFH⁺08, Cot10, FMP11, RBST14, Sin07] that go beyond the compression techniques we discussed earlier. One such technique is *RecycleUnits* [BIFH⁺08]: unit clauses in the proof are used to replace some clauses in the proof that are subsumed by the units. The replacement typically makes the proof invalid (i.e., some resolvents are no longer the result of resolving the antecedents). However, the proof can be fixed with a few simple

steps. Figure 15.7 illustrates the RecycleUnits procedure on a derivation of the example CNF formula E from page 638. In the example, notice that replacing a clause by a unit may strengthen the resolvent.

Two other proof compression techniques are *LowerUnits* and *RecyclePivots*. LowerUnits uses the observation that one needs to resolve over a unit clause only a single time. In case a unit clause occurs multiple times in a proof, it is lowered to ensure that it occurs only once. RecyclePivots reduces the *irregularity* in resolution proofs. A proof is irregular if it contains a path on which resolution is performed on the same pivot. Removing all irregularity in a proof may result in an exponential blow-up of the proof [Goe90]. Hence, techniques such as RecyclePivots need to be restricted in the context of proof compression. The tool *Skeptic* [BFP14], which includes most of the compression techniques, can be used to remove redundancy from resolution proofs.

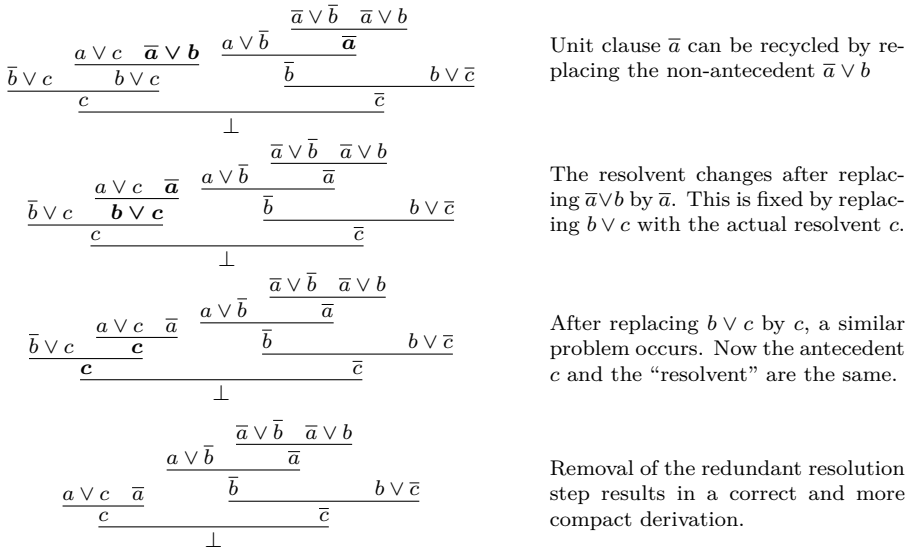


Figure 15.7. An example of the proof compression technique RecycleUnits.

15.6.2. Proofs Without Hints

Proofs without hints are checked using unit propagation. The actual check for each step depends on the proof system. Recall that a RUP proof $\{C_1, \dots, C_m\}$ is valid for formula F , if $C_m = \perp$ and for $i \in \{1, \dots, m\}$, it holds that

$$F \wedge C_1 \wedge \dots \wedge C_{i-1} \vdash C_i.$$

The check for stronger proof systems is more elaborate and thus more expensive. See Section 15.2 for details. The most simple (but very costly) method to validate proofs without hints checks for every $i \in \{1, \dots, m\}$ whether or not the redundancy property of the corresponding proof system holds.

One can reduce the cost of validating proofs without hints by checking them *backwards* [GN03], thereby *marking* clauses that are used for the derivation of other clauses that occur later in the proof: Initially, only the final clause $C_m = \perp$ is marked. Now we loop over the lemmas in backwards order, i.e., C_m, \dots, C_1 . Before validating a lemma, we first check whether it is marked. If a lemma is not marked, it can be skipped, which reduces the computational cost. If a lemma is marked, we check whether the clause satisfies the above redundancy criterion. If the check fails, the proof is invalid. Otherwise, we mark all clauses that were required to make the check succeed (using conflict analysis). For most proofs, this technique allows to skip over half of the lemmas during validation.

The main challenge regarding the validation of proofs without hints is efficiency: Validating a proof without hints is typically much more expensive than obtaining the proof using a SAT solver, even if the implementation uses backwards checking and the same data structures as state-of-the-art solvers. For most other logics, including first order logic, checking a proof is typically cheaper than finding the proof in the first place. There are two main reasons why checking unsatisfiability proofs is more expensive than solving.

First, SAT solvers aggressively delete clauses during solving, which reduces the cost of unit propagation. If the proof checker has no access to the clause deletion information, then unit propagation is much more expensive in the checker than in the solver. This was the main motivation why the proof formats DRUP and DRAT have been developed. These formats support the inclusion of clause-deletion information, thereby making the unit propagation costs between the solver and checker similar.

Second, SAT solvers *reuse* propagations between conflicts whereas proof checkers do not reuse propagations. Consider two consecutive lemmas C_i and C_{i+1} , produced by a SAT solver. In the most extreme (but not unusual) case, the branch that resulted in C_i and C_{i+1} differs only in a single decision (out of many decisions). Hence most propagations are reused by the solver. At the same time, the lemmas might have no overlapping literals, meaning that $C_i \cap C_{i+1} = \emptyset$. Consequently, the checker would not be able to reuse any propagations. In case $C_i \cap C_{i+1}$ is non-empty, the checker could potentially reuse propagations, although no checker implementation for proofs without hints exploits this.

While checking a proof without hints, one can easily produce an unsatisfiable core and a proof with hints. The unsatisfiable core consists of the original clauses that were marked during backwards checking. For most unsatisfiable formulas that arise from applications, many clauses are unmarked and thus not part of the unsatisfiable core. The proof with hints contains for each marked lemma all the clauses that were required during its validation as hints.

The hints that are generated while checking a proof without hints might differ significantly from the hints that would have been produced by the SAT solver. For example, the resolution proof (with hints) for the example formula E might be equal to the resolution graph shown in Fig. 15.1. On the other hand, the resolution proof produced by checking the proofs without hints might be equal to Fig. 15.8. Notice that the resolution graph of Fig. 15.8 (right) does not use all original clauses. Clause $(\bar{b} \vee c)$ is redundant and not part of the core of E .

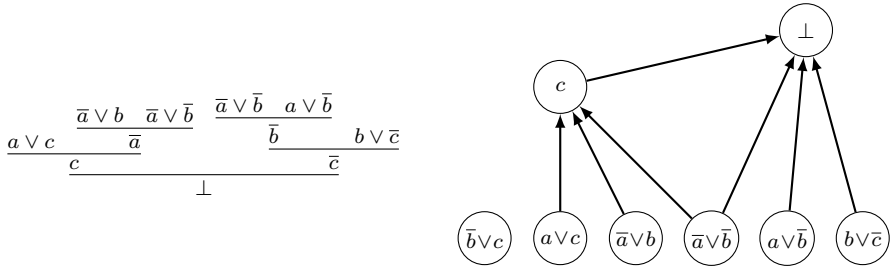


Figure 15.8. A resolution derivation (left) and a resolution graph (right) for the example formula E produced by checking a proof without hints.

15.7. Proof Applications

Proofs of unsatisfiability have been used to validate the results of SAT competitions.² Initially, during the SAT competitions of 2007, 2009, and 2011, a special track was organized for which the unsatisfiability results were checked. For the SAT competitions of 2013 and 2014, proof logging became mandatory for tracks with only unsatisfiable benchmarks. The supported formats for the SAT competition 2013 were `TraceCheck` and `DRUP`, but all solvers participating in these tracks opted for the `DRUP` format. For the SAT competition 2014, the only supported format was `DRAT`, which is backwards compatible with `DRUP`. Since 2016, the SAT competition requires proof logging to participate in the main track.

As already mentioned, one motivation for using proofs is to make testing and debugging of SAT solvers more effective. Checking learned clauses online with RUP allows to localize unsound implementation defects as soon they lead to clauses that are not implied by reverse unit propagation.

Testing with forward proof checking is particularly effective in combination with fuzzing (generating easy formulas) and delta-debugging [BB09b] (shrinking a formula that triggers a bug). Otherwise failures produced by unsound reasoning can only be observed if they turn a satisfiable instance into an unsatisfiable one. This situation is not only difficult to produce, but also tends to lead to much larger input files after delta-debugging.

However, model-based testing [ABS13] of the incremental API of a SAT solver is in our experience at least as effective as file-based fuzzing and delta-debugging. The online proof checking capabilities to `Lingeling` [Bie14] allow the combination of these two methods (model-based testing and proof checking).

Another important aspect of proof tracing is that it allows to generate a clausal (or variable) core (i.e., an unsatisfiable subset). These cores can be used in many applications, including MUS extraction [NRS13], MaxSAT [MHL⁺13], diagnosis [SKK03, NBE12], and abstraction refinement in model checking [EMA10] or SMT [ALS06, BB09a]. Note that this list of references is subjective and by far not complete. It should only be considered as a starting point for investigating related work on using cores.

²see <http://www.satcompetition.org> for details.

Finally, extraction of interpolants is an important usage of resolution proofs, particularly in the context of interpolation-based model checking [McM03]. Since resolution proofs are large and not easy to obtain, there have been several recent attempts to avoid proofs and obtain interpolants directly, see for instance [VRN13]. Interpolation-based model checking became the state-of-the-art until the invention of IC3 [Bra11]. The IC3 algorithm is also based on SAT technology, and also uses cores, but usually in a much more light-weight way. Typical implementations use assumption-based core techniques as introduced in MiniSAT [ES03] (see also [LB13]) instead of proof-based techniques.

15.8. Conclusions

Unsatisfiability proofs are useful for several applications, such as computing interpolants and MUS extraction. These proofs can also be used to validate results of the SAT solvers that produced them and for tools that use SAT solvers, such as theorem provers.

Proofs of unsatisfiability come in two types: with and without hints. Proofs without hints are useful for various applications, but they are costly and complicated to produce. Therefore few SAT solvers support proof logging with hints. Proof logging without hints is easy and therefore most state-of-the-art solvers support it. However, validating proofs without hints is costly, although recent advances significantly improved performance of checkers.

There are several challenges regarding unsatisfiability proofs. How can one store proofs with hints using much less space on disk and produce them using much less memory overhead? Can the cost of validating proofs with hints be reduced further? Last but not least, research is required to study how some techniques, such as Gaussian elimination, cardinality resolution, and symmetry breaking, can be expressed elegantly in unsatisfiability proofs.

Acknowledgements

The author thanks Benjamin Kiesl, Emre Yolcu, and David Narvaez for their useful comments on earlier drafts. This chapter reuses parts from “Proofs for Satisfiability Problem”, the first chapter of “All about Proofs, Proofs for all” [HB15].

References

- [ABS13] Cyrille Artho, Armin Biere, and Martina Seidl. Model-based testing for verification back-ends. In Margus Veanes and Luca Viganò, editors, *TAP*, volume 7942 of *Lecture Notes in Computer Science*, pages 39–55. Springer, 2013.
- [AFG⁺11] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. Verifying SAT and SMT in Coq for a fully automated decision procedure. In *International Workshop on Proof-Search in Axiomatic Theories and Type Theories (PSATTT)*, 2011.

- [ALS06] Zaher S. Andraus, Mark H. Liffiton, and Karem A. Sakallah. Refinement strategies for verification methods based on datapath abstraction. In *Proc. ASP-DAC'06*, pages 19–24. IEEE, 2006.
- [Amj07] Hasan Amjad. Compressing propositional refutations. *Electr. Notes Theor. Comput. Sci.*, 185:3–15, 2007.
- [ANORC08] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Efficient generation of unsatisfiability proofs and cores in SAT. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *LPAR*, volume 5330 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2008.
- [AS13] Gilles Audemard and Laurent Simon. Glucose 2.3 in the SAT 2013 Competition. In Anton Belov, Marijn J. H. Heule, and Matti Järvisalo, editors, *Proceedings of SAT Competition 2013*, volume B-2013-1 of *Department of Computer Science Series of Publications B, University of Helsinki*, pages 42–43, 2013.
- [ASM06] Fadi A. Aloul, Karem A. Sakallah, and Igor L. Markov. Efficient symmetry breaking for Boolean satisfiability. *IEEE Trans. Computers*, 55(5):549–558, 2006.
- [BB09a] Robert Brummayer and Armin Biere. Effective bit-width and under-approximation. In *Proc. EUROCAST'09*, volume 5717 of *LNCS*, pages 304–311, 2009.
- [BB09b] Robert Brummayer and Armin Biere. Fuzzing and delta-debugging SMT solvers. In *International Workshop on Satisfiability Modulo Theories (SMT)*, pages 1–5. ACM, 2009.
- [BFP14] Joseph Boudou, Andreas Fellner, and Bruno Woltzenlogel Paleo. Skeptik: A proof compression system. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings*, volume 8562 of *Lecture Notes in Computer Science*, pages 374–380. Springer, 2014.
- [BHMN20] Joshua Brakensiek, Marijn J. H. Heule, John Mackey, and David Narváez. The resolution of keller’s conjecture. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning*, pages 48–65, Cham, 2020. Springer International Publishing.
- [BHMS14] Anton Belov, Marijn J. H. Heule, and Joao P. Marques-Silva. Mus extraction using clausal proofs. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing — SAT 2014*, volume 8561 of *Lecture Notes in Computer Science*, pages 48–57. Springer International Publishing, 2014.
- [Bie07] Armin Biere. The AIGER and-inverter graph (AIG) format, version 20070427, 2007.
- [Bie08] Armin Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 4:75–97, 2008.
- [Bie13] Armin Biere. Lingeling, Plingeling and Treengeling entering the SAT Competition 2013. In Anton Belov, Marijn J. H. Heule, and Matti Järvisalo, editors, *Proceedings of SAT Competition 2013*, volume B-

- 2013-1 of *Department of Computer Science Series of Publications B, University of Helsinki*, pages 51–52, 2013.
- [Bie14] Armin Biere. Yet another local search solver and Lingeling and friends entering the SAT Competition 2014. In Anton Belov, Marijn J. H. Heule, and Matti Järvisalo, editors, *SAT Competition 2014*, volume B-2014-2 of *Department of Computer Science Series of Publications B*, pages 39–40. University of Helsinki, 2014.
- [BIFH⁺08] Omer Bar-Ilan, Oded Fuhrmann, Shlomo Hoory, Ohad Shacham, and Ofer Strichman. Linear-time reductions of resolution proofs. In Hana Chockler and Alan J. Hu, editors, *Haifa Verification Conference*, volume 5394 of *Lecture Notes in Computer Science*, pages 114–128. Springer, 2008.
- [BKS04] Paul Beame, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *JAIR*, 22:319–351, 2004.
- [BLB10] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 6175 of *LNCS*, pages 44–57. Springer, 2010.
- [Bra11] Aaron R. Bradley. SAT-based model checking without unrolling. In Ranjit Jhala and David A. Schmidt, editors, *VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011.
- [BS10] Roderick Bloem and Natasha Sharygina, editors. *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*. IEEE, 2010.
- [BT19] Sam Buss and Neil Thapen. DRAT proofs, propagation redundancy, and extended resolution. In Mikolás Janota and Inês Lynce, editors, *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, volume 11628 of *Lecture Notes in Computer Science*, pages 71–89. Springer, 2019.
- [BW03] Fahiem Bacchus and Jonathan Winter. Effective preprocessing with hyper-resolution and equality reduction. In Giunchiglia and Tacchella [GT04], pages 341–355.
- [CCT87] W. Cook, C. R. Coullard, and G. Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25 – 38, 1987.
- [CFHH⁺17] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt, Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified rat verification. In Leonardo de Moura, editor, *Automated Deduction – CADE 26*, pages 220–236, Cham, 2017. Springer International Publishing.
- [Coo76] Stephen A. Cook. A short proof of the pigeon hole principle using extended resolution. *SIGACT News*, 8(4):28–32, October 1976.
- [Cot10] Scott Cotton. Two techniques for minimizing resolution proofs. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing — SAT 2010*, volume 6175 of *Lecture Notes*

- in *Computer Science*, pages 306–312. Springer, 2010.
- [CR79] Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *The Journal of Symbolic Logic*, 44(1):pp. 36–50, 1979.
- [DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.
- [EB05] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.
- [EMA10] Niklas Eén, Alan Mishchenko, and Nina Amla. A single-instance incremental SAT formulation of proof- and counterexample-based abstraction. In Bloem and Sharygina [BS10], pages 181–188.
- [ES03] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Giunchiglia and Tacchella [GT04], pages 502–518.
- [FMP11] Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Compression of propositional resolution proofs via partial regularization. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wrocław, Poland, July 31 - August 5, 2011. Proceedings*, volume 6803 of *Lecture Notes in Computer Science*, pages 237–251. Springer, 2011.
- [GN03] Evguenii I. Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 10886–10891. IEEE, 2003.
- [Goe90] Andreas Goerdt. Comparing the complexity of regular and unrestricted resolution. In Heinz Marburger, editor, *GWAI-90 14th German Workshop on Artificial Intelligence*, volume 251 of *Informatik-Fachberichte*, pages 181–185. Springer Berlin Heidelberg, 1990.
- [GT04] Enrico Giunchiglia and Armando Tacchella, editors. *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*. Springer, 2004.
- [GV14] Arie Gurfinkel and Yakir Vizel. Druping for interpolants. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, FMCAD '14, pages 19:99–19:106, Austin, TX, 2014. FMCAD Inc.
- [Hak85] Armin Haken. The intractability of resolution. *Theor. Comput. Sci.*, 39:297–308, 1985.
- [HB15] Marijn J.H. Heule and Armin Biere. Proofs for satisfiability problems. In Bruno Woltzenlogel Paleo and David Delahaye, editors, *All about Proofs, Proofs for all*, chapter 1. College Publications, 2015.
- [HB16] Marijn J. H. Heule and Armin Biere. Clausal proof compression. In Boris Konev, Stephan Schulz, and Laurent Simon, editors, *IWIL-2015. 11th International Workshop on the Implementation of Logics*,

- volume 40 of *EPiC Series in Computing*, pages 21–26. EasyChair, 2016.
- [HB18] Marijn J. H. Heule and Armin Biere. What a difference a variable makes. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 75–92, Cham, 2018. Springer International Publishing.
- [Heu18] Marijn J. H. Heule. Schur number five. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18)*, pages 6598–6606. AAAI Press, 2018.
- [HHJW15] Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Expressing symmetry breaking in DRAT proofs. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, pages 591–606, Cham, 2015. Springer International Publishing.
- [HHKW17] Marijn J. H. Heule, Warren Hunt, Matt Kaufmann, and Nathan Wetzler. Efficient, verified checking of propositional proofs. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving*, pages 269–284, Cham, 2017. Springer International Publishing.
- [HHW13a] Marijn J. H. Heule, Warren A. Hunt, Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 181–188. IEEE, 2013.
- [HHW13b] Marijn J. H. Heule, Warren A. Hunt, Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In *International Conference on Automated Deduction (CADE)*, volume 7898 of *LNAI*, pages 345–359. Springer, 2013.
- [HKB19] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Strong extension-free proof systems. *Journal of Automated Reasoning*, 64:533 – 554, 2019.
- [HKM16] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing - SAT 2016*, pages 228–245, Cham, 2016. Springer International Publishing.
- [HKS17] Marijn J. H. Heule, Benjamin Kiesl, Martina Seidl, and Armin Biere. Pruning through satisfaction. In Ofer Strichman and Rachel Tzoref-Brill, editors, *Hardware and Software: Verification and Testing*, pages 179–194, Cham, 2017. Springer International Publishing.
- [HKWB11] Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *HVC*, volume 7261 of *LNCS*, pages 50–65. Springer, 2011.
- [HS09] HyoJung Han and Fabio Somenzi. On-the-fly clause improvement. In Kullmann [Kul09], pages 209–222.
- [HSB14] Marijn J. H. Heule, Martina Seidl, and Armin Biere. A unified proof system for qbf preprocessing. In Stéphane Demri, Deepak Kapur,

- and Christoph Weidenbach, editors, *Automated Reasoning*, pages 91–106, Cham, 2014. Springer International Publishing.
- [JBH12] Matti Järvisalo, Armin Biere, and Marijn Heule. Simulating circuit-level simplifications on CNF. *J. Autom. Reasoning*, 49(4):583–619, 2012.
- [JHB12] Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. Inprocessing rules. In *International Joint Conference on Automated Reasoning (IJCAR)*, volume 7364 of *LNCS*, pages 355–370. Springer, 2012.
- [KHB19] Benjamin Kiesl, Marijn J. H. Heule, and Armin Biere. Truth assignments as conditional autarkies. In Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza, editors, *Automated Technology for Verification and Analysis*, pages 48–64, Cham, 2019. Springer International Publishing.
- [KL14] Boris Konev and Alexei Lisitsa. A SAT attack on the erdős discrepancy conjecture. pages 219–226, 2014.
- [KRPH18] Benjamin Kiesl, Adrián Rebola-Pardo, and Marijn J. H. Heule. Extended resolution simulates DRAT. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *Automated Reasoning*, pages 516–531, Cham, 2018. Springer International Publishing.
- [KSTB16] Benjamin Kiesl, Martina Seidl, Hans Tompits, and Armin Biere. Super-blocked clauses. In Nicola Olivetti and Ashish Tiwari, editors, *Automated Reasoning*, pages 45–61, Cham, 2016. Springer International Publishing.
- [Kul99] Oliver Kullmann. On a generalization of extended resolution. *Discrete Applied Mathematics*, 96-97:149–176, 1999.
- [Kul09] Oliver Kullmann, editor. *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*. Springer, 2009.
- [Lam20] Peter Lammich. Efficient verified (UN)SAT certificate checking. *Journal of Automated Reasoning*, 64:513–532, 2020.
- [LB13] Jean-Marie Lagniez and Armin Biere. Factoring out assumptions to speed up MUS extraction. In Matti Järvisalo and Allen Van Gelder, editors, *SAT*, volume 7962 of *Lecture Notes in Computer Science*, pages 276–292. Springer, 2013.
- [McM03] Kenneth L. McMillan. Interpolation and SAT-based model checking. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.
- [MHB12] Norbert Manthey, Marijn J. H. Heule, and Armin Biere. Automated reencoding of Boolean formulas. In *Proceedings of Haifa Verification Conference (HVC)*, volume 6397 of *LNCS*, pages 102–117. Springer, 2012.
- [MHL⁺13] António Morgado, Federico Heras, Mark H. Liffiton, Jordi Planes, and Joao Marques-Silva. Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints*, 18(4):478–534, 2013.
- [Nad10] Alexander Nadel. Boosting minimal unsatisfiable core extraction.

- In Bloem and Sharygina [BS10], pages 221–229.
- [NBE12] Alexander Nöhrer, Armin Biere, and Alexander Egyed. Managing SAT inconsistencies with HUMUS. In *Proc. VaMoS'12*, pages 83–91. ACM, 2012.
- [NRS13] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Efficient MUS extraction with resolution. In *FMCAD*, pages 197–200. IEEE, 2013.
- [RBST14] Simone Fulvio Rollini, Roberto Bruttomesso, Natasha Sharygina, and Aliaksei Tsitovich. Resolution proof transformation for compression and interpolation. *Formal Methods in System Design*, 45(1):1–41, 2014.
- [Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.
- [SB06] Carsten Sinz and Armin Biere. Extended resolution proofs for conjoining bdds. In Dima Grigoriev, John Harrison, and Edward A. Hirsch, editors, *CSR*, volume 3967 of *Lecture Notes in Computer Science*, pages 600–611. Springer, 2006.
- [SB09] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In Kullmann [Kul09], pages 237–243.
- [Sin07] Carsten Sinz. Compressing propositional proofs by common subproof extraction. In Roberto Moreno-Díaz, Franz Pichler, and Alexis Quesada-Arencibia, editors, *EUROCAST*, volume 4739 of *Lecture Notes in Computer Science*, pages 547–555. Springer, 2007.
- [SKK03] Carsten Sinz, Andreas Kaiser, and Wolfgang Küchlin. Formal methods for the validation of automotive product configuration data. *Artif. Intell. Eng. Des. Anal. Manuf.*, 17(1):75–97, January 2003.
- [Soo13] Mate Soos. Strangeness. In A. Belov, M. Heule, and M. Järvisalo, editors, *Proceedings of SAT Competition 2013*, volume B-2013-1 of *Department of Computer Science Series of Publications B, University of Helsinki*, pages 89–90, 2013.
- [Tse83] Grigori S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of Reasoning 2*, pages 466–483. Springer, 1983.
- [Urq87] Alasdair Urquhart. Hard examples for resolution. *J. ACM*, 34(1):209–219, 1987.
- [Van08] Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *International Symposium on Artificial Intelligence and Mathematics (ISAIM)*, 2008.
- [Van09] Allen Van Gelder. Improved conflict-clause minimization leads to improved propositional proof traces. In Kullmann [Kul09], pages 141–146.
- [Van12] Allen Van Gelder. Producing and verifying extremely large propositional refutations - have your cake and eat it too. *Ann. Math. Artif. Intell.*, 65(4):329–372, 2012.
- [VRN13] Yakir Vizel, Vadim Ryvchin, and Alexander Nadel. Efficient generation of small interpolants in CNF. In Natasha Sharygina and Helmut Veith, editors, *CAV*, volume 8044 of *Lecture Notes in Com-*

- puter Science*, pages 330–346. Springer, 2013.
- [WA09] Tjark Weber and Hasan Amjad. Efficiently checking propositional refutations in HOL theorem provers. *Journal of Applied Logic*, 7(1):26–40, 2009.
- [Web06] Tjark Weber. Efficiently checking propositional resolution proofs in Isabelle/HOL. In *International Workshop on the Implementation of Logics (IWIL)*, volume 212, pages 44–62, 2006.
- [WHH13] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt, Jr. Mechanical verification of SAT refutations with extended resolution. In *Conference on Interactive Theorem Proving (ITP)*, volume 7998 of *LNCS*, pages 229–244. Springer, 2013.
- [WHH14] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt, Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *Theory and Applications of Satisfiability Testing (SAT)*, 2014.
- [WMB99] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes (2Nd Ed.): Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 1999.
- [ZM03] Lintao Zhang and Sharad Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1, DATE '03*, pages 10880–10885. IEEE Computer Society, 2003.

Chapter 16

Worst-Case Upper Bounds

Evgeny Dantsin and Edward A. Hirsch

There are many algorithms for testing satisfiability — how to evaluate and compare them? It is common (but still disputable) to identify the efficiency of an algorithm with its worst-case complexity. From this point of view, asymptotic upper bounds on the worst-case running time and space is a criterion for evaluation and comparison of algorithms. In this chapter we survey ideas and techniques behind satisfiability algorithms with the currently best upper bounds. We also discuss some related questions: “easy” and “hard” cases of SAT, reducibility between various restricted cases of SAT, the possibility of solving SAT in subexponential time, etc.

In Section 16.1 we define terminology and notation used throughout the chapter. Section 16.2 addresses the question of which special cases of SAT are polynomial-time tractable and which ones remain **NP**-complete. The first non-trivial upper bounds for testing satisfiability were obtained for algorithms that solve k -SAT; such algorithms also form the core of general SAT algorithms. Section 16.3 surveys the currently fastest algorithms for k -SAT. Section 16.4 shows how to use bounds for k -SAT to obtain the currently best bounds for SAT. Section 16.5 addresses structural questions like “what else happens if k -SAT is solvable in time $\langle \dots \rangle$?”. Finally, Section 16.6 summarizes the currently best bounds for the main cases of the satisfiability problem.

16.1. Preliminaries

16.1.1. Definitions and notation

A *literal* is a Boolean variable or its negation. A *clause* is a finite set of literals that does not contain a variable together with its negation. By a *formula* we mean a Boolean formula in conjunctive normal form (*CNF formula*) defined as a finite set of clauses. The number of literals in a clause is called the *length* of the clause. A formula F is called a k -*CNF* formula if every clause in F has length at most k .

Throughout the chapter we write n , m , and l to denote the following natural parameters of a formula F :

- n is the number of variables occurring in F ;
- m is the number of clauses in F ;
- l is the total number of occurrences of all variables in F .

We also write $|F|$ to denote the length of a reasonable binary representation of F , i.e., the size of the input in the usual complexity-theoretic sense. The ratio m/n is called the *clause density* of F .

An *assignment* is a mapping from a set of variables to $\{\text{true}, \text{false}\}$. We identify an assignment with a set A of literals: if a variable x is mapped to **true** then $x \in A$; if x is mapped to **false** then $\neg x \in A$. Given a formula F and an assignment A , we write $F[A]$ to denote the result of substitution of the truth values assigned by A . Namely, $F[A]$ is a formula obtained from F as follows:

- all clauses that contain literals belonging to A are removed from F ;
- all literals that are complementary to the literals in A are deleted from the remaining clauses.

For example, for $F = \{\{x, y\}, \{\neg y, z\}\}$ and $A = \{y\}$, the formula $F[A]$ is $\{z\}$. If C is a clause, we write $F[-C]$ to denote $F[A]$ where A consists of the literals complementary to the literals in C .

An assignment A *satisfies* a formula F if $F[A] = \emptyset$. If a formula has a satisfying assignment, it is called *satisfiable*.

SAT denotes the language of all satisfiable CNF formulas (we sometimes refer to this language as *General SAT* to distinguish it from its restricted versions), *k-SAT* is its subset consisting of k -CNF formulas, *SAT- f* is the subset of SAT containing formulas with each variable occurring at most f times, *k-SAT- f* is the intersection of the last two languages. *Unique k-SAT* denotes a promise problem: an algorithm solves the problem if it gives correct answers for k -CNF formulas that have at most one satisfying assignment (and there are no requirements on the behavior of the algorithm on other formulas).

Complexity bounds are given using the standard asymptotic notation (O , Ω , o , etc), see for example [CLRS01]. We use \log to denote the binary logarithm and \ln to denote the natural logarithm. The binary entropy function is given by

$$H(x) = -x \log x - (1 - x) \log(1 - x).$$

16.1.2. Transformation rules

We describe several simple operations that transform formulas without changing their satisfiability status (we delay more specific operations until they are needed).

Unit clauses. A clause is called *unit* if its length is 1. Clearly, a unit clause determines the truth value of its variable in a satisfying assignment (if any). Therefore, all unit clauses can be eliminated by the corresponding substitutions. This procedure (iterated until no unit clauses are left) is known as *unit clause elimination*.

Subsumption. Another simple procedure is *subsumption*: if a formula contains two clauses and one is a subset of the other, then the larger one can be dropped.

Resolution. If two clauses A and B have exactly one pair of complementary literals $a \in A$ and $\neg a \in B$, then the clause $A \cup B \setminus \{a, \neg a\}$ is called the *resolvent* of A and B (by a) and denoted by $R(A, B)$. The resolvent is a logical consequence of these clauses and can be added to the formula without changing its satisfiability. The important cases of adding resolvents are:

- *Resolution with subsumption.* If F contains two clauses C and D such that their resolvent $R(C, D)$ is a subset of D , replace F by $(F \setminus \{D\}) \cup \{R(C, D)\}$.
- *Elimination of a variable by resolution.* Given a formula F and a literal a , the formula denoted $DP_a(F)$ is constructed from F by adding all resolvents by a and then removing all clauses that contain a or $\neg a$ (this transformation was used in [DP60]). A typical use of this rule is as follows: if the number of clauses (resp. the number of literal occurrences) in $DP_a(F)$ is smaller than in F , replace F by $DP_a(F)$.
- *Bounded resolution.* Add all resolvents of size bounded by some function of the formula parameters.

16.2. Tractable and intractable classes

Consider the satisfiability problem for a certain class of formulas. Depending on the class, either this restriction is in \mathbf{P} (2-SAT for example) or no polynomial-time algorithm for the restricted problem is known (3-SAT). Are there any criteria that would allow us to distinguish between tractable and intractable classes? A remarkable result in this direction was obtained by Schaefer in 1978 [Sch78]. He considered a generalized version of the satisfiability problem, namely he considered Boolean constraint satisfaction problems $\text{SAT}(\mathcal{C})$ where \mathcal{C} is a set of constraints, see precise definitions below. Each set \mathcal{C} determines a class of formulas such as 2-CNF, 3-CNF, Horn formulas, etc. Loosely speaking, Schaefer's dichotomy theorem states that there are only two possible cases: $\text{SAT}(\mathcal{C})$ is either in \mathbf{P} or \mathbf{NP} -complete. The problem is in \mathbf{P} if and only if the class determined by \mathcal{C} has at least one of the following properties:

1. Every formula in the class is satisfied by assigning true to all variables;
2. Every formula in the class is satisfied by assigning false to all variables;
3. Every formula in the class is a Horn formula;
4. Every formula in the class is a dual-Horn formula;
5. Every formula in the class is in 2-CNF;
6. Every formula in the class is affine.

We describe classes with these properties in the next section. Schaefer's theorem and some relevant results are given in Section 16.2.2.

16.2.1. "Easy" classes

Trivially satisfiable formulas. A CNF formula F is satisfied by assigning true (resp. false) to all variables if and only if every clause in F has at least one positive (resp. negative) literal. The satisfiability problem for such formulas is trivial: all formulas are satisfiable.

Horn and dual-Horn formulas. A CNF formula is called *Horn* if every clause in this formula has at most one positive literal. Satisfiability of a Horn formula F can be tested as follows. If F has unit clauses then we apply unit clause elimination until all unit clauses are eliminated. If the resulting formula F' contains the empty clause, we conclude that F is unsatisfiable. Otherwise, every clause in F' contains at least two literals and at least one of them is negative. Therefore, F' is trivially satisfiable by assigning false to all variables, which means that F is satisfiable too. It is obvious that this method takes polynomial time. Using data flow techniques, satisfiability of Horn formulas can be tested in linear time [DG84].

A CNF formula is called *dual-Horn* if every clause in it has at most one negative literal. The satisfiability problem for dual-Horn formulas can be solved similarly to the case of Horn formulas.

2-CNF formulas. A linear-time algorithm for 2-SAT was given in [APT79]. This algorithm represents an input formula F by a labeled directed graph G as follows. The set of vertices of G consists of $2n$ vertices corresponding to all variables occurring in F and their negations. Each clause $a \vee b$ in F can be viewed as two implications $\neg a \rightarrow b$ and $\neg b \rightarrow a$. These two implications produce two edges in the graph: $(\neg a, b)$ and $(\neg b, a)$. It is easy to prove that F is unsatisfiable if and only if there is a variable x such that G has a cycle containing both x and $\neg x$. Since strongly connected components can be computed in linear time, 2-SAT can be solved in linear time too.

There are also other methods of solving 2-SAT in polynomial time, for example, random walks [Pap94] or resolution (note that the resolution rule applied to 2-clauses produces a 2-clause, so the number of all possible resolvents does not exceed $O(n^2)$).

Affine formulas. A *linear equation over the two-element field* is an expression of the form $x_1 \oplus \dots \oplus x_k = \delta$ where \oplus denotes the sum modulo 2 and δ stands for 0 or 1. Such an equation can be expressed as a CNF formula consisting of 2^{k-1} clauses of length k . An *affine formula* is a conjunction of linear equations over the two-element field. Using Gaussian elimination, we can test satisfiability of affine formulas in polynomial time.

16.2.2. Schaefer's dichotomy theorem

Definition 16.2.1. A function $\phi: \{\text{true}, \text{false}\}^k \rightarrow \{\text{true}, \text{false}\}$ is called a *Boolean constraint* of arity k .

Definition 16.2.2. Let ϕ be a constraint of arity k and x_1, \dots, x_k be a sequence of Boolean variables (possibly with repetitions). The pair $\langle \phi, (x_1, \dots, x_k) \rangle$ is called a *constraint application*. Such a pair is typically written as $\phi(x_1, \dots, x_k)$.

Definition 16.2.3. Let $\phi(x_1, \dots, x_k)$ be a constraint application and A be a truth assignment to x_1, \dots, x_k . We say that A *satisfies* $\phi(x_1, \dots, x_k)$ if ϕ evaluates to true on the truth values assigned by A .

Definition 16.2.4. Let $\Phi = \{C_1, \dots, C_m\}$ be a set of constraint applications and A be a truth assignment to all variables occurring in Φ . We say that A is a *satisfying assignment* for Φ if A satisfies every constraint application in Φ .

Definition 16.2.5. Let \mathcal{C} be a set of Boolean constraints. We define $\text{SAT}(\mathcal{C})$ to be the following decision problem: given a finite set Φ of applications of constraints from \mathcal{C} , is there a satisfying assignment for Φ ?

Example. Let \mathcal{C} consist of four Boolean constraints $\phi_1, \phi_2, \phi_3, \phi_4$ defined as follows:

$$\begin{aligned}\phi_1(x, y) &= x \vee y; \\ \phi_2(x, y) &= x \vee \neg y; \\ \phi_3(x, y) &= \neg x \vee \neg y; \\ \phi_4(x, y) &= \text{false}.\end{aligned}$$

Then $\text{SAT}(\mathcal{C})$ is 2-SAT.

Theorem 16.2.1 (Schaefer’s dichotomy theorem [Sch78]). *Let \mathcal{C} be a set of Boolean constraints. If \mathcal{C} satisfies at least one of the conditions (1)–(6) below then $\text{SAT}(\mathcal{C})$ is in \mathbf{P} . Otherwise $\text{SAT}(\mathcal{C})$ is \mathbf{NP} -complete.*

1. Every constraint in \mathcal{C} evaluates to true if all arguments are true.
2. Every constraint in \mathcal{C} evaluates to true if all arguments are false.
3. Every constraint in \mathcal{C} can be expressed as a Horn formula.
4. Every constraint in \mathcal{C} can be expressed as a dual-Horn formula.
5. Every constraint in \mathcal{C} can be expressed as a 2-CNF formula.
6. Every constraint in \mathcal{C} can be expressed as an affine formula.

The main part of this theorem is to prove that $\text{SAT}(\mathcal{C})$ is \mathbf{NP} -hard if none of the conditions (1)–(6) holds for \mathcal{C} . The proof is based on a classification of the sets of constraints that are closed under certain operations (conjunction, substitution of constants, existential quantification). Any such set either satisfies one of the conditions (3)–(6) or coincides with the set of all constraints.

Schaefer’s theorem was extended and refined in many directions, for example a complexity classification of the polynomial-time solvable case (with respect to \mathbf{AC}^0 reducibility) is given in [ABI⁺05]. Likewise, classification theorems similar to Schaefer’s one were proved for many variants of SAT, including the counting satisfiability problem #SAT, the quantified satisfiability problem QSAT, the maximum satisfiability problem MAX SAT (see Part 4 of this book for the definitions of these extensions of SAT), and others, see a survey in [CKS01].

A natural question arises: given a set \mathcal{C} of constraints, how difficult is it to recognize whether $\text{SAT}(\mathcal{C})$ is “easy” or “hard”? That is, we are interested in the complexity of the following “meta-problem”: Given \mathcal{C} , is $\text{SAT}(\mathcal{C})$ in \mathbf{P} ? As shown in [CKS01], the complexity of the meta-problem depends on how the constraints are specified:

- If each constraint in \mathcal{C} is specified by its set of satisfying assignments then the meta-problem is in \mathbf{P} .
- If each constraint in \mathcal{C} is specified by a CNF formula then the meta-problem is **co-NP**-hard.

- If each constraint in \mathcal{C} is specified by a DNF formula then the meta-problem is **co-NP-hard**.

16.3. Upper bounds for k -SAT

SAT can be trivially solved in 2^n polynomial-time steps by trying all possible assignments. The design of better exponential-time algorithms started with papers [MS79, Dan81, Luc84, MS85] exploiting a simple divide-and-conquer approach¹ for k -SAT. Indeed, if a formula contains a 3-clause $\{x_1, x_2, x_3\}$, then deciding its satisfiability is equivalent to deciding the satisfiability of three simpler formulas $F[x_1]$, $F[\neg x_1, x_2]$, and $F[\neg x_1, \neg x_2, x_3]$, and the recurrent inequality on the number of leaves in such a computation tree already gives a $|F|^{O(1)} \cdot 1.84^n$ -time algorithm for 3-SAT, see Part 1, Chapter 8 for a survey of techniques for estimating the running time of such algorithms.

Later development brought more complicated algorithms based on this approach both for k -SAT [Kul99] and SAT [KL97, Hir00], see also Section 16.4.2. However, modern randomized algorithms and their derandomized versions appeared to give better bounds for k -SAT. In what follows we survey these new approaches: *critical clauses* (Section 16.3.1), *multistart random walks* (Section 16.3.2), and *cube covering* proposed in an attempt to derandomize the random-walk technique (Section 16.3.3). Finally, we go briefly through recent improvements for particular cases (Section 16.3.4).

16.3.1. Critical clauses

To give the idea of the next algorithm, let us switch for a moment to a particular case of k -SAT where a formula has at most one satisfying assignment (Unique k -SAT). The restriction of this case ensures that every variable v must occur in a v -critical clause that is satisfied only by the literal corresponding to v (otherwise one could change its value to obtain a different satisfying assignment); we call v the *principal* variable of this clause. This observation remains also valid for general k -SAT if we consider an *isolated* satisfying assignment that becomes unsatisfying if we flip the value of exactly one variable. More generally, we call a satisfying assignment j -isolated if this happens for exactly j variables (while for any other of the remaining $n - j$ variables, a flip of its value yields another satisfying assignment). Clearly, each of these j variables has its critical clause, i.e., a clause where it is the principal variable.

Now start picking variables one by one at random and assigning random values to them eliminating unit clauses after each step. Clearly, each critical clause of length k has a chance of $1/k$ of being “properly” ordered: its principal variable is chosen after all other variables in the clause and thus will be assigned “for free” during the unit clause elimination provided the values of other variables are chosen correctly. Thus the expected number of values we get “for free” is j/k ; the probability to guess correctly the values of other variables is $2^{-(n - \lfloor j/k \rfloor)}$.

¹Such SAT algorithms are usually called *DPLL algorithms* since the approach originates from [DP60, DLL62].

While this probability of success is the largest for 1-isolated satisfying assignments, j -isolated assignments for larger j come in larger “bunches”. The overall probability of success is

$$\sum_j 2^{-(n-\lfloor j/k \rfloor)} \cdot (\text{number of } j\text{-isolated assignments}),$$

which can be shown to be $\Omega(2^{-n(1-1/k)})$ [PPZ97]. The probability that the number of “complimentary” values is close to its expectation adds an inverse polynomial factor. Repeating the procedure in order to get a constant probability of error yields an $|F|^{O(1)} \cdot 2^{n(1-1/k)}$ -time algorithm, which can be derandomized for some penalty in time.

The above approach based on critical clauses was proposed in [PPZ97]. A follow-up [PPSZ98] (journal version: [PPSZ05]) gives an improvement of the $2^{n(1-1/k)}$ bound by using a preprocessing step that augments the formula by all resolvents of size bounded by a slowly growing function. Adding the resolvents formalizes the intuition that a critical clause may become “properly ordered” not only in the original formula, but also during the process of substituting the values.

Algorithm 16.3.1 (The PPSZ algorithm, [PPSZ05]).

Input: k -CNF formula F .

Output: yes, if F is satisfiable, no otherwise.²

1. Augment F using bounded resolution up to the length bounded by any slowly growing function such that $s(n) = o(\log n)$ and s tends to infinity as n tends to infinity.
2. Repeat $2^{n(1-\frac{\mu_k}{k-1}+o(1))}$ times, where $\mu_k = \sum_{j=1}^{\infty} \frac{1}{j(j+\frac{1}{k-1})}$:
 - (a) Apply unit clause elimination to F .
 - (b) Pick randomly a literal a still occurring in F and replace F by $F[a]$.³

Theorem 16.3.1 ([PPSZ05]). *The PPSZ algorithm solves k -SAT in time*

$$|F|^{O(1)} \cdot 2^{n(1-\frac{\mu_k}{k-1}+o(1))}$$

with error probability $o(1)$, where μ_k is an increasing sequence with $\mu_3 \approx 1.227$ and $\lim_{k \rightarrow \infty} \mu_k = \pi^2/6$.

16.3.2. Multistart random walks

A typical local-search algorithm for SAT starts with an initial assignment and modifies it step by step, trying to come closer to a satisfying assignment (if any). Methods of modifying may vary. For example, greedy algorithms choose a variable and flip its value so as to increase some function of the assignment, say,

²This algorithm and the other randomized algorithms described below may incorrectly say “no” on satisfiable formulas.

³Contrary to [PPSZ05], we choose a literal not out of all $2n$ possibilities (for n variables still remaining in the formula) but from a possibly smaller set. However, this only excludes a chance to assign the wrong value to a pure literal.

the number of satisfied clauses. Another method was used in Papadimitriou's algorithm for 2-SAT [Pap91]: flip the value of a variable chosen at random from an unsatisfied clause. Algorithms based on this method are referred to as *random-walk* algorithms.

Algorithm 16.3.2 (The multistart random-walk algorithm).

Input: k -CNF formula F .

Output: satisfying assignment A if F is satisfiable, no otherwise.

Parameters: integers t and w .

1. Repeat t times:
 - (a) Choose an assignment A uniformly at random.
 - (b) If A satisfies F , return A and halt. Otherwise repeat the following instructions w times:
 - Pick any unsatisfied clause C in F .
 - Choose a variable x uniformly at random from the variables occurring in C .
 - Modify A by flipping the value of x in A .
 - If the updated assignment A satisfies F , return A and halt.
2. Return no and halt.

This multistart random-walk algorithm with $t = 1$ and $w = 2n^2$ is Papadimitriou's polynomial-time algorithm for 2-SAT [Pap91]. The algorithm with $t = (2 - 2/k)^n$ and $w = 3n$ is known as Schöning's algorithm for k -SAT where $k \geq 3$ [Sch99].

Random-walk algorithms can be analyzed using one-dimensional random walks. Consider one random walk. Suppose that the input formula F has a satisfying assignment S that differs from the current assignment A in the values of exactly j variables. At each step of the walk, A becomes closer to S with probability at least $1/k$ because any unsatisfied clause contains at least one variable whose values in A and in S are different. Thus, the performance of the algorithm can be described in terms of a particle walking on the integer interval $[0..n]$. The position 0 corresponds to the satisfying assignment S . At each step, if the particle's position is j , where $0 < j < n$, it moves to $j - 1$ with probability at least $1/k$ and moves to $j + 1$ with probability at most $1 - 1/k$.

Let p be the probability that a random walk that starts from a random assignment finds a fixed satisfying assignment S in at most $3n$ steps. Using the "Ballot theorem", Schöning shows in [Sch99] that $p \geq (2 - 2/k)^{-n}$ up to a polynomial factor. An alternative analysis by Welzl reduces the polynomial factor to a constant: $p \geq (2/3) \cdot (2 - 2/k)^{-n}$, see Appendix to [Sch02]. Therefore, Schöning's algorithm solves k -SAT with the following upper bound:

Theorem 16.3.2 ([Sch02]). *Schöning's algorithm solves k -SAT in time*

$$|F|^{O(1)} \cdot (2 - 2/k)^n$$

with error probability $o(1)$.

$$\underbrace{\left\{ \begin{array}{c} \text{word}_1 \\ \text{word}_2 \\ \dots \\ \text{word}_N \end{array} \right\} \cdot \left\{ \begin{array}{c} \text{word}_1 \\ \text{word}_2 \\ \dots \\ \text{word}_N \end{array} \right\} \cdot \left\{ \begin{array}{c} \text{word}_1 \\ \text{word}_2 \\ \dots \\ \text{word}_N \end{array} \right\} \dots \left\{ \begin{array}{c} \text{word}_1 \\ \text{word}_2 \\ \dots \\ \text{word}_N \end{array} \right\}}_t$$

Figure 16.1. A block code C^t for code $C = \{\text{word}_1, \dots, \text{word}_N\}$.

16.3.3. Cube covering

Schöning’s algorithm solves k -SAT using an exponential number of relatively “short” random walks that start from random assignments. Is it possible to derandomize this approach? A derandomized version of multistart random walks is given in [DGH⁺02].

The idea behind the derandomization can be described in terms of a covering of the Boolean cube with Hamming balls. We can view truth assignments to n variables as points in the Boolean cube $\{0, 1\}^n$. For any two points $u, v \in \{0, 1\}^n$, the *Hamming distance* between u and v is denoted by $\delta(u, v)$ and defined to be the number of coordinates in which these points are different. The *Hamming ball* of radius R with center w is defined to be

$$\{u \in \{0, 1\}^n \mid \delta(u, w) \leq R\}.$$

Let \mathcal{C} be a collection of Hamming balls of radius R with centers w_1, \dots, w_t . This collection is called a *covering* of the cube if each point in the cube belongs to at least one ball from \mathcal{C} . When thinking of the centers w_1, \dots, w_t as *codewords*, we identify the covering \mathcal{C} with a *covering code* [CHLL97].

The derandomization of Schöning’s algorithm for k -SAT is based on the following approach. First, we cover the Boolean cube with Hamming balls of some fixed radius R (two methods of covering are described below). This covering should contain as few balls as possible. Then we consider every ball and search for a satisfying assignment inside it (see the procedure *Search* below). The time of searching inside a ball depends on the radius R : the smaller R we use, the less time is needed. However, if the radius is small, many balls are required to cover the cube. It is shown in [DGH⁺02] that for the particular procedure *Search* the overall running time is minimum when $R = n/(k + 1)$.

Covering of the Boolean cube with Hamming balls. How many Hamming balls of a given radius R are required to cover the cube? Since the volume of a Hamming ball of radius R is at most $2^{nH(R/n)}$ where H is the binary entropy function, the smallest possible number of balls is at least $2^{n(1-H(R/n))}$. This trivial lower bound is known as *sphere covering bound* [CHLL97]. Our task is to construct a covering code whose size is close to the sphere covering bound. To do it, we partition n bits into d blocks with n/d bits in each block. Then we construct a covering code of radius R/d for each block and, finally, we define a covering code for the n bits to be the direct sum of the covering codes for blocks (see Fig. 16.1). More exactly, the cube-covering-based algorithm in [DGH⁺02] uses the following two methods of covering.

- *Method A: Constant number of blocks of linear size.* We partition n bits into d blocks where d is a constant. Then we use a greedy algorithm for the Set Cover problem to construct a covering code for a block with n/d bits. The resulting covering code for n bits is optimal: its size achieves the sphere covering bound up to a polynomial factor. However, this method has a disadvantage: construction of a covering code for a block requires exponential space.
- *Method B: Linear number of blocks of constant size.* We partition n bits into n/b blocks of size b each, where b is a constant. A covering code for each block is constructed using a brute-force method. The resulting covering code is constructed within polynomial space, however this code is only “almost” optimal: its size achieves the sphere covering bound up to a factor of $2^{\varepsilon n}$ for arbitrary $\varepsilon > 0$.

Search inside a ball. After constructing a covering of the cube with balls, we search for a satisfying assignment inside every ball. To do this, we build a tree using the local search paradigm: if the current assignment A does not satisfy the input formula, choose any unsatisfied clause $\{l_1, \dots, l_k\}$ and consider assignments A_1, \dots, A_k where A_i is obtained from A by flipping the literal l_i (note that while a traditional randomized local search considers just *one* of these assignments, a deterministic procedure must consider *all* these assignments). More exactly, we use the following recursive procedure $Search(F, A, R)$. It takes as input a k -CNF formula F , an assignment A (center of the ball), and a number R (radius of the ball). If F has a satisfying assignment inside the ball, the procedure outputs yes (for simplicity, we describe a decision version); otherwise, the answer is no.

1. If F is true under A , return yes.
2. If $R \leq 0$, return no.
3. If F contains the empty clause, return no.
4. Choose any clause that is false under A . For each literal l in C , run $Search(F[l], A, R - 1)$. Return yes if at least one of these calls returns yes. Otherwise return no.

The procedure builds a tree in which

- each degree is at most k ;
- the height is at most R (the height is the distance from the center of the ball).

Therefore, $Search(F, A, R)$ runs in time k^R up to a polynomial factor.

Algorithm 16.3.3 (The cube-covering-based algorithm, [DGH⁺02]).

Input: k -CNF formula F .

Output: yes if F is satisfiable, no otherwise.

Parameter: $\varepsilon > 0$ (the parameter is needed only if Method B is used).

1. Cover the Boolean cube with Hamming balls of radius $R = n/(k+1)$ using either Method A or Method B.
2. For each ball, run $Search(F, A, R)$ where A is the center of the ball. Return yes if at least one of these calls returns yes. Otherwise, return false.

Theorem 16.3.3 ([DGH⁺02]). *The cube-covering-based algorithm solves k -SAT in time*

$$|F|^{O(1)} \cdot (2 - 2/(k+1))^n$$

and within exponential space if Method A is used, or in time

$$|F|^{O(1)} \cdot (2 - 2/(k+1) + \varepsilon)^n$$

and within polynomial space if Method B is used.

In the next section we refer to a weakened version of these bounds:

Proposition 16.3.4. *The cube-covering-based algorithm solves k -SAT in time*

$$|F|^{O(1)} \cdot 2^{n(1-1/k)}.$$

16.3.4. Improvements for restricted versions of k -SAT

3-SAT. Like the general case of k -SAT, the currently best bounds for randomized 3-SAT algorithms are better than those for deterministic 3-SAT algorithms.

- *Randomized algorithms.* Schöning's algorithm solves 3-SAT formulas in time $O(1.334^n)$. It was shown in [IT04] that this bound can be improved by combining Schöning's algorithm with the PPSZ algorithm. The currently best bound obtained using such a combination is $O(1.323^n)$ [Rol06].
- *Deterministic algorithms.* The cube-covering-based algorithm solves 3-SAT in time $O(1.5^n)$. This bound can be improved using a pruning technique for search inside a ball [DGH⁺02]. The currently best bound based on this method is $O(1.473^n)$ [BK04].

Unique k -SAT. The currently best bounds for Unique k -SAT are obtained using the approach based on critical clauses.

- *Randomized algorithms.* For $k > 4$, the best known bound for Unique k -SAT is the bound given by the PPSZ algorithm (Theorem 16.3.1). For Unique 3-SAT and Unique 4-SAT, the PPSZ bounds can be improved: $O(1.308^n)$ and $O(1.470^n)$ respectively [PPSZ05].
- *Deterministic algorithms.* The PPSZ algorithms can be derandomized for Unique k -SAT using the technique of limited independence [Rol05]. This derandomization yields bounds that can be made arbitrarily close to the bounds for the randomized case (see above) by taking appropriate values for the parameters of the derandomized version.

16.4. Upper bounds for General SAT

Algorithms with the currently best upper bounds for General SAT are based on the following two approaches:

- **Clause shortening** (Section 16.4.1). This method gives the currently best bound of the form $|F|^{O(1)} \cdot 2^{\alpha n}$ where α depends on n and m .
- **Branching** (Section 16.4.2, see also Part 1, Chapter 8). Algorithms based on branching have the currently best upper bounds of the form $|F|^{O(1)} \cdot 2^{\beta m}$ and $|F|^{O(1)} \cdot 2^{\gamma l}$, where β and γ are constants.

16.4.1. Algorithms based on clause shortening

The first nontrivial upper bound for SAT was given by Pudlak in [Pud98]. His randomized algorithm used the critical clauses method [PPZ97] to solve SAT with the $|F|^{O(1)} \cdot 2^{\alpha n}$ bound where $\alpha = 1 - 1/(2\sqrt{n})$. A slightly worse bound was obtained using a deterministic algorithm based on the cube-covering technique [DHW04].

Further better bounds were obtained using the *clause-shortening* approach proposed by Schuler in [Sch05]. This approach is based on the following dichotomy: *For any “long” clause (longer than some $k = k(n, m)$), either we can shorten this clause by choosing any k literals in the clause and dropping the other literals, or we can substitute false for these k literals in the entire formula.*

In other words, if a formula F contains a “long” clause C then F is equivalent to $F' \vee F''$ where F' is obtained from F by shortening C to a subclause D of length k and F'' is obtained from F by assigning false to all literals in D . This dichotomy is used to shorten all “long” clauses in the input formula F and apply a k -SAT algorithm afterwards.

The clause-shortening approach was described in [Sch05] as a randomized algorithm. Its derandomized version was given in [DHW06].

Algorithm 16.4.1 (The clause-shortening algorithm).

Input: CNF formula F consisting of clauses C_1, \dots, C_m .

Output: yes if F is satisfiable, no otherwise.

Parameter: integer k .

1. Change each clause C_i to a clause D_i as follows: If $|C_i| > k$ then choose any k literals in C_i and drop the other literals; otherwise leave C_i as is. Let F' denote the resulting formula $D_1 \vee \dots \vee D_m$.
2. Test satisfiability of F' using the cube-covering-based algorithm (Section 16.3).
3. If F' is satisfiable, output yes and halt. Otherwise, for each i , do the following:
 - (a) Convert F to F_i as follows:
 - i. Replace C_j by D_j for all $j < i$;
 - ii. Assign false to all literals in D_i .
 - (b) Recursively invoke this algorithm on F_i .
4. Return no.

What value of k is optimal? If k is small then each k -SAT instance can be solved fast, however we have a large number of k -SAT instances. If k is large then the k -SAT subroutine takes much time. Schuler’s bound and further better bounds for the clause-shortening approach were improved by Calabro, Impagliazzo, and Paturi in [CIP06]. Using this approach, they reduce a SAT instance to an exponential number of k -SAT instances and prove the following “conditional” bound:

Theorem 16.4.1 ([CIP06]). *If k -SAT can be solved in time $|F|^{O(1)} \cdot 2^{\alpha n}$ then SAT can be solved in time*

$$|F|^{O(1)} \cdot 2^{\alpha n + \frac{4m}{2^{\alpha k}}}$$

for any n, m and k such that $m \geq n/k$.

Although this theorem is not used in [CIP06] to give an improved bound for SAT explicitly, such a bound can be easily derived as follows. Since there is an algorithm that solves k -SAT with the $2^{n(1-1/k)}$ bound (Corollary 16.3.4), we can take $\alpha = 1 - 1/k$. Now we would like to choose k such as to minimize the exponent

$$\alpha n + \frac{4m}{2^{\alpha k}}$$

in the bound given by Theorem 16.4.1 above. Assuming that $m > n$, we take $k = 2 \log(m/n) + c$ where c is a constant that will be chosen later. Then

$$\begin{aligned} \alpha n + \frac{4m}{2^{\alpha k}} &= n \left(1 - \frac{1}{k}\right) + \frac{4m}{2^{k-1}} \\ &= n \left(1 - \frac{1}{2 \log(m/n) + c} + \frac{1}{(m/n) 2^{c-3}}\right) \end{aligned}$$

The constant c can be chosen so that

$$2 \log(m/n) + c < (m/n) 2^{c-3}$$

for all $m > n$. Therefore, the exponent is bounded by $n(1 - 1/O(k))$, which gives us the currently best upper bound for SAT:

Proposition 16.4.2. *SAT can be solved (for example, by Algorithm 16.4.1) in time*

$$|F|^{O(1)} \cdot 2^{n(1 - \frac{1}{O(\log(m/n))})}$$

where $m > n$.

16.4.2. Branching-based algorithms

Branching heuristics (aka DPLL algorithms) gave the first nontrivial upper bounds for k -SAT in the form $|F|^{O(1)} \cdot 2^{\alpha n}$ where α depends on k (Section 16.3). This approach was also used to obtain the first nontrivial upper bounds for SAT as functions of other parameters on input formulas: Monien and Speckenmeyer proved the $|F|^{O(1)} \cdot 2^{m/3}$ bound [MS80]; Kullmann and Luckhardt proved the $|F|^{O(1)} \cdot 2^{l/9}$ bound [KL97]. Both these bounds are improved by Hirsch in [Hir00]:

Theorem 16.4.3 ([Hir00]). *SAT can be solved in time*

1. $|F|^{O(1)} \cdot 2^{0.30897 m}$;
2. $|F|^{O(1)} \cdot 2^{0.10299 l}$.

To prove these bounds, two branching-based algorithms are built. They split the input formula F into either two formulas $F[x], F[\neg x]$ or four formulas $F[x, y], F[x, \neg y], F[\neg x, y], F[\neg x, \neg y]$. Then the following transformation rules are used (Section 16.1.2):

- *Elimination of unit clauses.*

- *Subsumption.*
- *Resolution with subsumption.*
- *Elimination of variables by resolution.* The rule is being applied as long as it does not increase m (resp., l).
- *Elimination of blocked clauses.* Given a formula F , a clause C , and a literal a in C , we say that C is *blocked* [Kul99] for a with respect to F if the literal $\neg a$ occurs only in those clauses of F that contain the negation of at least one of the literals occurring in $C \setminus \{a\}$. In other words, C is blocked for a if there are no resolvents by a of the clause C and any other clause in F . Given F and a , we define the assignment $I(a, F)$ to be consisting of the literal a plus all other literals b of F such that
 - b is different from a and $\neg a$;
 - the clause $\{ \neg a, b \}$ is blocked for $\neg a$ with respect to F .
 The following two facts about blocked clauses are proved by Kullmann in [Kul99]:
 - If C is blocked for a with respect to F , then F and $F \setminus \{C\}$ are equisatisfiable.
 - For any literal a , the formula F is satisfiable iff at least one of the formulas $F[\neg a]$ and $F[I(a, F)]$ is satisfiable.
 The rule is based on the first statement: if C is blocked for a with respect to F , replace F by $F \setminus \{C\}$.
- *Black and white literals.* This rule [Hir00] generalizes the obvious fact that if all clauses in F contain the same literal then F is satisfiable. Let P be a binary relation between literals and formulas such that for a variable v and a formula F , at most one of $P(v, F)$ and $P(\neg v, F)$ holds. Suppose that each clause in F that contains a “white” literal w satisfying $P(w, F)$ also contains a “black” literal b satisfying $P(\neg b, F)$. Then, for any literal a such that $P(\neg a, F)$, the formula F can be replaced by $F[a]$.

Remark. The second bound is improved to $2^{0.0926l}$ using another branching-based algorithm [Wah05].

Formulas with constant clause density. Satisfiability of formulas with constant clause density ($m/n \leq \text{const}$) can be solved in time $O(2^{\alpha n})$ where $\alpha < 1$ and α depends on the clause density [AS03, Wah05]. This fact also follows from Theorem 16.4.3 for $m \leq n$ and from Corollary 16.4.2 for $m > n$. The latter also shows how α grows as m/n increases:

$$\alpha = 1 - \frac{1}{O(\log(m/n))}.$$

16.5. How large is the exponent?

Given a number of $2^{\alpha n}$ -time algorithms for various variants of SAT, it is natural to ask how the exponents αn are related to each other (and how they are related to exponents for other combinatorial problems of similar complexity). Another natural question is whether α can be made arbitrarily small.

Recall that a language in **NP** can be identified with a polynomially balanced relation $R(x, y)$ that checks that y is indeed a solution to the instance x . (We

refer the reader to [Pap94] or some other handbook for complexity theory notions such as polynomially balanced relation or oracle computation.)

Definition 16.5.1. *Parameterized NP problem* (L, q) consists of a language $L \in \mathbf{NP}$ (defined by a polynomially-time computable and polynomially balanced relation R such that $x \in L \iff \exists y R(x, y)$) and a complexity parameter q , which is a polynomial-time computable⁴ integer function that bounds the size of the shortest solution to R : $x \in L \iff \exists y (|y| \leq q(x) \wedge R(x, y))$.

For $L = \text{SAT}$ or $k\text{-SAT}$, natural complexity parameters studied in the literature are the number n of variables, the number m of clauses, and the total number l of literal occurrences. We will be interested whether these problems can be solved in time bounded by a *subexponential* function of the parameter according to the following definition.

Definition 16.5.2 ([IPZ01]). A parameterized problem $(L, q) \in \mathbf{SE}$ if for every positive integer k there is a deterministic Turing machine deciding the membership problem $x \in L$ in time $|x|^{O(1)} \cdot 2^{q(x)/k}$.

Our ultimate goal is then to figure out which versions of the satisfiability problem belong to \mathbf{SE} . An *Exponential Time Hypothesis (ETH)* for a problem (L, q) says that $(L, q) \notin \mathbf{SE}$. A natural conjecture saying that the search space for $k\text{-SAT}$ must be necessarily exponential in the number of variables is $(3\text{-SAT}, n) \notin \mathbf{SE}$.

It turns out, however, that the only thing we can hope for is a kind of completeness result relating the complexity of SAT to the complexity of other combinatorial problems. The completeness result needs the notion of a reduction. However, usual polynomial-time reductions are not tight enough for this. Even linear-time reductions are not always sufficient. Moreover, it is by far not straightforward even that ETH for $(k\text{-SAT}, n)$ is equivalent to ETH for $(k\text{-SAT}, m)$, see Section 16.5.1 for a detailed treatment of these questions.

Even if one assumes ETH for 3-SAT to be true, it is not clear how the exponents for different versions of $k\text{-SAT}$ are related to each other, see Section 16.5.2.

16.5.1. SERF-completeness

The first attempt to investigate systematically ETH for $k\text{-SAT}$ has been made by Impagliazzo, Paturi, and Zane [IPZ01] who introduced reductions that preserve the fact that the complexity of the parameterized problem is subexponential.

Definition 16.5.3 ([IPZ01]). A series $\{T_k\}_{k \in \mathbb{N}}$ of oracle Turing machines forms a *subexponential reduction family (SERF)* reducing parameterized problem (A, p) to problem (B, q) if T_k^B solves the problem $F \in A$ in time $|F|^{O(1)} \cdot 2^{p(F)/k}$ and queries its oracle for instances $I \in B$ with $q(I) = O(p(F))$ and $|I| = |F|^{O(1)}$. (The constants in $O(\cdot)$ do not depend on k .)

⁴Note that the definition is different from the one being used in the parameterized complexity theory (Chapter 17).

It is easy to see that SERF reductions are transitive and preserve subexponential time. In particular, once a SERF-complete problem for a class of parameterized problems is established, the existence of subexponential-time algorithms for the complete problem is equivalent to the existence of subexponential time algorithms for every problem in the class. It turns out that the class **SNP** defined by Kolaitis and Vardi [KV87] (cf. [PY91]) indeed has SERF-complete problems.

Definition 16.5.4 (parameterized version of **SNP** from [IPZ01]). A parameterized problem $(L, q) \in \mathbf{SNP}$ if L can be defined by a series of second-order existential quantifiers followed by a series of first-order universal quantifiers followed by a quantifier-free first-order formula in the language of the quantified variables and the quantified relations f_1, \dots, f_s and input relations h_1, \dots, h_u (given by tables of their values on a specific universum of size n):

$$(h_1, \dots, h_u) \in L \iff \exists f_1, \dots, f_s \forall p_1, \dots, p_t \Phi(p_1, \dots, p_t),$$

where Φ uses f_i 's and h_i 's. The parameter q is defined as the number of bits needed to describe all relations f_i 's, i.e., $\sum_i n^{\alpha_i}$, where α_i is the arity of f_i .

Theorem 16.5.1 ([IPZ01]). $(3\text{-SAT}, m)$ and $(3\text{-SAT}, n)$ are SERF-complete for **SNP**.

SERF-completeness of SAT with the parameter n follows straightforwardly from the definitions (actually, the reduction queries only k -SAT instances with k depending on the problem to reduce). In order to reduce $(k\text{-SAT}, n)$ to $(k\text{-SAT}, m)$, one needs a *sparsification procedure* which is interesting in its own right. Then the final reduction of $(k\text{-SAT}, m)$ to $(3\text{-SAT}, m)$ is straightforward.

Algorithm 16.5.5 (The sparsification procedure [CIP06]).

Input: k -CNF formula F .

Output: sequence of k -CNF formulas that contains a satisfiable formula if and only if F is satisfiable.

Parameters: integers θ_i for $i \geq 0$.

1. For $c = 1, \dots, n$, for $h = c, \dots, 1$, search F for $t \geq \theta_{c-h}$ clauses $C_1, C_2, \dots, C_t \in F$ of length c such that $|\bigcap_{i=1}^t C_i| = h$. Stop after the first such set of clauses is found and let $H = \bigcap_{i=1}^t C_i$. If nothing is found, output F and exit.
2. Make a recursive call for $F \setminus \{C_1, \dots, C_t\} \cup \{H\}$.
3. Make a recursive call for $F[\neg H]$.

Calabro, Impagliazzo, and Paturi [CIP06] show that, given k and sufficiently small ε , for

$$\theta_i = O\left(\left(\frac{k^2}{\varepsilon} \log \frac{k}{\varepsilon}\right)^{i+1}\right)$$

this algorithm outputs at most $2^{\varepsilon n}$ formulas such that

- for every generated formula, every variable occurs in it at most $(k/\varepsilon)^{3k}$ times;

- F is satisfiable if and only if at least one of the generated formulas is satisfiable.

Since the number of clauses in the generated formulas is bounded by a linear function of the number of variables in F , Algorithm 16.5.5 defines a SERF reduction from $(k\text{-SAT}, n)$ to $(k\text{-SAT}, m)$.

16.5.2. Relations between the exponents

The fact of SERF-completeness, though an important advance in theory, does not give per se the exact relation between the exponents for different versions of $k\text{-SAT}$ if one assumes ETH for these problems. Let us denote the exponents for the most studied versions of $k\text{-SAT}$ (here $\mathbf{RTime}[t(n)]$ denotes the class of problems solvable in time $O(t(n))$ by a randomized one-sided bounded error algorithm):

$$\begin{aligned}
 s_k &= \inf\{\delta \geq 0 \mid k\text{-SAT} \in \mathbf{RTime}[2^{\delta n}]\}, \\
 s_k^{\text{freq.}f} &= \inf\{\delta \geq 0 \mid k\text{-SAT-}f \in \mathbf{RTime}[2^{\delta n}]\}, \\
 s_k^{\text{dens.}d} &= \inf\{\delta \geq 0 \mid k\text{-SAT with at most } dn \text{ clauses} \in \mathbf{RTime}[2^{\delta n}]\}, \\
 s_k^{\text{freq.}f} &= \inf\{\delta \geq 0 \mid \text{SAT-}f \in \mathbf{RTime}[2^{\delta n}]\}, \\
 s_k^{\text{dens.}d} &= \inf\{\delta \geq 0 \mid \text{SAT with at most } dn \text{ clauses} \in \mathbf{RTime}[2^{\delta n}]\}, \\
 \sigma_k &= \inf\{\delta \geq 0 \mid \text{Unique } k\text{-SAT} \in \mathbf{RTime}[2^{\delta n}]\}, \\
 s_\infty &= \lim_{k \rightarrow \infty} s_k, \\
 s^{\text{freq.}\infty} &= \lim_{f \rightarrow \infty} s^{\text{freq.}f}, \\
 s^{\text{dens.}\infty} &= \lim_{d \rightarrow \infty} s^{\text{dens.}d}, \\
 \sigma_\infty &= \lim_{k \rightarrow \infty} \sigma_k.
 \end{aligned}$$

In the remaining part of this section we survey the dependencies between these numbers. Note that we are interested here in randomized (as opposed to deterministic) algorithms, because some of our reductions are randomized.

$k\text{-SAT}$ vs Unique $k\text{-SAT}$

Valiant and Vazirani [VV86] presented a randomized polynomial-time reduction of SAT to its instances having at most one satisfying assignment (Unique SAT). The reduction, however, neither preserves the maximum length of a clause nor is capable to preserve subexponential complexity.

As a partial progress in the above problem, it is shown in [CIKP03] that $s_\infty = \sigma_\infty$.

Lemma 16.5.2 ([CIKP03]). $\forall k \forall \varepsilon \in (0, \frac{1}{4}) s_k \leq \sigma_{k'} + O(H(\varepsilon))$, where $k' = \max\{k, \frac{1}{\varepsilon} \ln \frac{2}{\varepsilon}\}$.

Proposition 16.5.3 ([CIKP03]). $s_k \leq \sigma_k + O(\frac{\ln^2 k}{k})$. Thus $s_\infty = \sigma_\infty$.

In order to prove Lemma 16.5.2 [CIKP03] employs an *isolation procedure*, which is a replacement for the result of [VV86]. The procedure starts with *concentrating* the satisfying assignments of the input formula in a Hamming ball of small radius εn . This is done by adding a bunch of “size k' ” random hyperplanes $\bigoplus_{i \in R} a_i x_i = b$ (encoded in CNF by listing the $2^{k'-1}$ k' -clauses that are implied by this equality), where the subset R of size k' and bits a_i, b are chosen at random, to the formula. For $k' = \max\{k, \frac{1}{\varepsilon} \ln \frac{2}{\varepsilon}\}$ and a linear number of hyperplanes, this yields a “concentrated” k' -CNF formula with substantial probability of success.

Once the assignments are concentrated, it remains to isolate a random one. To do that, the procedure guesses a random set of variables in order to approximate (from the above) the set of variables having different values in different satisfying assignments. Then the procedure guesses a correct assignment for the chosen set. Both things are relatively easy to guess, because the set is small.

k -SAT for different values of k

Is the sequence s_3, s_4, \dots of the k -SAT complexities an increasing sequence (as one may expect from the definition)? Impagliazzo and Paturi prove in [IP01] that the sequence s_k is strictly increasing infinitely often. More exactly, if there exists $k_0 \geq 3$ such that ETH is true for k_0 -SAT, then for every k there is $k' > k$ such that $s_k < s_{k'}$. This result follows from the theorem below, which can be proved using both critical clauses and sparsification techniques.

Theorem 16.5.4 ([IP01]). $s_k \leq (1 - \Omega(k^{-1}))s_\infty$.

Note that [IP01] gives an explicit constant in $\Omega(k^{-1})$.

k -SAT vs SAT- f

The sparsification procedure gives immediately

$$s_k \leq s_k^{\text{freq.}((k/\varepsilon)^{3k})} + \varepsilon \leq s_k^{\text{dens.}((k/\varepsilon)^{3k})} + \varepsilon.$$

In particular, for $\varepsilon = k^{-O(1)}$ this means that k -SAT could be only slightly harder than (k) -SAT of exponential density, and $s_\infty \leq s^{\text{freq.}\infty} \leq s^{\text{dens.}\infty}$.

The opposite inequality is shown using the clause-shortening algorithm: substituting an $O(2^{s_k + \varepsilon n})$ -time k -SAT algorithm and $m \leq dn$ into Theorem 16.4.1 we get (after taking the limit as $\varepsilon \rightarrow 0$)

$$s_k^{\text{dens.}d} \leq s_k + \frac{4d}{2^{ks_k}}. \quad (16.1)$$

Choosing k as a function of d so that the last summand is smaller than the difference between s_k and s_∞ , [CIP06] shows that taking (16.1) to the limit gives $s^{\text{dens.}\infty} \leq s_\infty$, i.e.,

$$s_\infty = s^{\text{freq.}\infty} = s^{\text{dens.}\infty}.$$

16.6. Summary table

The table below summarizes the currently best upper bounds for 3-SAT, k -SAT for $k > 3$, and SAT with no restriction on clause length (the bounds are given up

to polynomial factors). Other “record” bounds are mentioned in previous sections, for example bounds for Unique k -SAT (Section 16.3.4), bounds as functions in m or l (Section 16.4.2), etc.

	randomized algorithms	deterministic algorithms
3-SAT	1.323^n [Rol06]	1.473^n [BK04]
k -SAT	$2^{n(1 - \frac{\mu k}{k-1} + o(1))}$ [PPSZ98]	$(2 - 2/(k+1))^n$ [DGH ⁺ 02]
SAT	$2^{n(1 - \frac{1}{\sigma(\log(m/n))})}$ [CIP06] ⁵	$2^{n(1 - \frac{1}{\sigma(\log(m/n))})}$ [CIP06] ⁵

16.7. Addendum for the 2nd Edition: Connections to Circuit Complexity

To prove an upper bound for a computational problem Π , it suffices to design a *single* algorithm that solves Π using resources (like time or space) that do not exceed this bound. To prove a lower bound for Π , one needs to show that *every* algorithm for solving Π requires a certain amount of resources. Ryan Williams showed in [Wil10, Wil11] that there is a nontrivial connection between these somewhat opposing tasks: a slight improvement over exhaustive search for Boolean satisfiability implies a nontrivial lower bound for circuit complexity.

In circuit complexity, the basic model of computation is a family $\{C_n\}_{n \in \mathbb{N}}$ where C_n is a circuit with n input bits. Such a family *decides* a language $L \subseteq \{0, 1\}^*$ if for every n -bit string x , the circuit C_n outputs 1 on x if and only if $x \in L$. Since there may be no single algorithm producing every circuit C_n , complexity classes defined in terms of circuits are commonly referred to as “non-uniform” classes in contrast to “uniform” complexity classes, like \mathbf{P} or \mathbf{NP} , defined in terms of algorithms.

The most natural example of a non-uniform class is $\mathbf{P/poly}$ that consists of all languages decidable by families $\{C_n\}_{n \in \mathbb{N}}$ where the size of C_n is polynomial in n . It is clear that each polynomial-time algorithm can be simulated using a family of polynomial-size circuits and, therefore, \mathbf{P} is a subset of $\mathbf{P/poly}$. What about nondeterministic polynomial-time algorithms, can they be simulated by non-uniform families of polynomial-size circuits? It is commonly believed that $\mathbf{NP} \not\subseteq \mathbf{P/poly}$ but the current state of complexity theory is far from proving this (such a proof would imply $\mathbf{P} \neq \mathbf{NP}$). Moreover, it is open whether much larger classes such as \mathbf{NEXP} or $\mathbf{E}^{\mathbf{NP}}$ are contained in $\mathbf{P/poly}$; the current best lower bound is $\mathbf{MAEXP} \not\subseteq \mathbf{P/poly}$ [BFT98].

A possible approach to proving lower bounds for non-uniform computations would be to restrict types of circuits. For example, \mathbf{ACC} denotes the class of languages decidable by constant-depth polynomial-size circuits with the following gates: NOT gates, AND gates and OR gates of unbounded fan-in, MOD- m gates of unbounded fan-in (such a gate outputs 0 if the number of ones in the input is divisible by m and outputs 1 otherwise). The question of strong lower bounds for \mathbf{ACC} was open until 2011 when Williams proved the bound $\mathbf{NEXP} \not\subseteq \mathbf{ACC}$

⁵This bound can be derived from Lemma 5 in [CIP06], see Corollary 16.4.2 above.

building on connections between upper bounds and lower bounds [Wil10, Wil11]. These exciting results can be summarized as follows.

Let \mathcal{C} be a class of circuits. For example, \mathcal{C} can consist of all polynomial-size circuits over AND, OR, NOT, or \mathcal{C} can be a class of circuits used in the definition of **ACC** above. Let \mathcal{C} -SAT denote the satisfiability problem for circuits of \mathcal{C} : given a circuit $C \in \mathcal{C}$, determine whether there is an input on which C outputs 1.

Theorem 16.7.1 ([Wil10, Wil11]). *Suppose there is a $c > 0$ such that \mathcal{C} -SAT can be solved on circuits with n inputs and n^k size in $O(2^n/n^c)$ time for every k . Then **NEXP** contains a language that cannot be decided by a family of polynomial-size circuits from the class \mathcal{C} .*

This theorem is proved by contradiction, assuming that \mathcal{C} -SAT can be solved by a $O(2^n/n^c)$ algorithm and **NEXP** has polynomial-size circuits from \mathcal{C} . These two assumptions together imply an algorithm that is “too good to exist”: this algorithm could be used to simulate every nondeterministic algorithm running in $O(2^n)$ time by a nondeterministic algorithm running in $o(2^n)$ time, which contradicts the nondeterministic hierarchy theorem [SFM78].

Theorem 16.7.2 ([Wil11]). *Let \mathcal{C} be the class of circuits with NOT gates and unbounded fan-in AND, OR, MOD- m gates (such circuits are used in the definition of the class **ACC**). For every integer $d > 0$, there is an $\epsilon \in (0, 1)$ and there is an algorithm that solves \mathcal{C} -SAT on circuits with n inputs of depth d and size at most 2^{n^ϵ} in 2^{n-n^ϵ} time.*

The proof of this theorem relies on simulating **ACC** circuits by low degree polynomials [Yao90, BT91].

Proposition 16.7.3 ([Wil11]). **NEXP** $\not\subseteq$ **ACC**.

There is also a bunch of results relating modest upper bounds for satisfiability to modest lower bounds for non-uniform classes. Here are two examples of such results.

Theorem 16.7.4 ([Wil10], Theorem 6.1). *If the Exponential Time Hypothesis is false, then **E^{NP}** does not have linear size circuits.*

Theorem 16.7.5 ([JMV18], Corollary 7). *If 3-SAT is in time c^n for any $c < 2^{1/7} = 1.10\dots$, then there exists a (non-Boolean) function $f: \{0, 1\}^n \rightarrow \{0, 1\}^2$ in **E^{NP}** such that any circuit over the full basis computing it requires at least $3n$ (non-input) gates.*

References

- [ABI⁺05] E. Allender, M. Bauland, N. Immerman, H. Schnoor, and H. Vollmer. The complexity of satisfiability problems: Refining Schaefer’s theorem. In *Proceedings of the 30th International Symposium on Mathematical Foundations of Computer Science, MFCS 2005*, volume 3618 of *Lecture Notes in Computer Science*, pages 71–82. Springer, 2005.

- [APT79] B. Aspvall, M. F. Plass, and R. E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–132, 1979.
- [AS03] V. Arvind and R. Schuler. The quantum query complexity of 0-1 knapsack and associated claw problems. In *Proceedings of the 14th Annual International Symposium on Algorithms and Computation, ISAAC 2003*, volume 2906 of *Lecture Notes in Computer Science*, pages 168–177. Springer, December 2003.
- [BFT98] H. Buhman, L. Fortnow, and T. Thierauf. Nonrelativizing separations. In *Proceedings of the 13th Annual IEEE Conference on Computational Complexity, Buffalo, New York, USA, June 15-18, 1998*, pages 8–12. IEEE Computer Society, 1998.
- [BK04] T. Brueggemann and W. Kern. An improved local search algorithm for 3-SAT. *Theoretical Computer Science*, 329(1–3):303–313, December 2004.
- [BT91] R. Beigel and J. Tarui. On ACC. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science, FOCS 1991*, pages 783–792, 1991. Journal version: *Computational Complexity* 4: 350–366 (1994).
- [CHLL97] G. Cohen, I. Honkala, S. Litsyn, and A. Lobstein. *Covering Codes*, volume 54 of *Mathematical Library*. Elsevier, Amsterdam, 1997.
- [CIKP03] C. Calabro, R. Impagliazzo, V. Kabanets, and R. Paturi. The complexity of unique k -SAT: An isolation lemma for k -CNFs. In *Proceedings of the 18th Annual IEEE Conference on Computational Complexity, CCC 2003*, pages 135–141. IEEE Computer Society, 2003.
- [CIP06] C. Calabro, R. Impagliazzo, and R. Paturi. A duality between clause width and clause density for SAT. In *Proceedings of the 21st Annual IEEE Conference on Computational Complexity, CCC 2006*, pages 252–260. IEEE Computer Society, 2006.
- [CKS01] N. Creignou, S. Khanna, and M. Sudan. *Complexity Classifications of Boolean Constraint Satisfaction Problems*. Society for Industrial and Applied Mathematics, 2001.
- [CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.
- [Dan81] E. Dantsin. Two propositional proof systems based on the splitting method. *Zapiski Nauchnykh Seminarov LOMI*, 105:24–44, 1981. In Russian. English translation: *Journal of Soviet Mathematics*, 22(3):1293–1305, 1983.
- [DG84] W. F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 1:267–284, 1984.
- [DGH⁺02] E. Dantsin, A. Goerdt, E. A. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan, and U. Schöning. A deterministic $(2 - 2/(k + 1))^n$ algorithm for k -SAT based on local search. *Theoretical Computer Science*, 289(1):69–83, October 2002.
- [DHW04] E. Dantsin, E. A. Hirsch, and A. Wolpert. Algorithms for SAT based on search in Hamming balls. In *Proceedings of the 21st Annual Sym-*

- posium on Theoretical Aspects of Computer Science, STACS 2004*, volume 2996 of *Lecture Notes in Computer Science*, pages 141–151. Springer, March 2004.
- [DHW06] E. Dantsin, E. A. Hirsch, and A. Wolpert. Clause shortening combined with pruning yields a new upper bound for deterministic SAT algorithms. In *Proceedings of the 6th Conference on Algorithms and Complexity, CIAC 2006*, volume 3998 of *Lecture Notes in Computer Science*, pages 60–68. Springer, May 2006.
- [DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.
- [DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.
- [Hir00] E. A. Hirsch. New worst-case upper bounds for SAT. *Journal of Automated Reasoning*, 24(4):397–420, 2000.
- [IP01] R. Impagliazzo and R. Paturi. On the complexity of k -SAT. *Journal of Computer and System Sciences*, 62(2):367–375, 2001.
- [IPZ01] R. Impagliazzo, R. Paturi, and F. Zane. Which problems have strongly exponential complexity. *Journal of Computer and System Sciences*, 63(4):512–530, 2001.
- [IT04] K. Iwama and S. Tamaki. Improved upper bounds for 3-SAT. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004*, page 328, January 2004.
- [JMV18] H. Jahanjou, E. Miles, and E. Viola. Local reduction. *Inf. Comput.*, 261(Part 2):281–295, 2018.
- [KL97] O. Kullmann and H. Luckhardt. Deciding propositional tautologies: Algorithms and their complexity. Technical report, Fachbereich Mathematik, Johann Wolfgang Goethe Universität, 1997.
- [Kul99] O. Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, 223(1–2):1–72, 1999.
- [KV87] P. G. Kolaitis and M. Y. Vardi. The decision problem for the probabilities of higher-order properties. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, STOC 1987*, pages 425–435. ACM, 1987.
- [Luc84] H. Luckhardt. Obere Komplexitätsschranken für TAUT-Entscheidungen. In *Proceedings of Frege Conference 1984, Schwerin*, pages 331–337. Akademie-Verlag Berlin, 1984.
- [MS79] B. Monien and E. Speckenmeyer. 3-satisfiability is testable in $O(1.62^n)$ steps. Technical Report Bericht Nr. 3/1979, Reihe Theoretische Informatik, Universität-Gesamthochschule-Paderborn, 1979.
- [MS80] B. Monien and E. Speckenmeyer. Upper bounds for covering problems. Technical Report Bericht Nr. 7/1980, Reihe Theoretische Informatik, Universität-Gesamthochschule-Paderborn, 1980.
- [MS85] B. Monien and E. Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics*, 10(3):287–295, March 1985.
- [Pap91] C. H. Papadimitriou. On selecting a satisfying truth assignment. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science, FOCS 1991*, pages 163–169, 1991.

- [Pap94] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
- [PPSZ98] R. Paturi, P. Pudlák, M. E. Saks, and F. Zane. An improved exponential-time algorithm for k -SAT. In *Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science, FOCS 1998*, pages 628–637, 1998.
- [PPSZ05] R. Paturi, P. Pudlák, M. E. Saks, and F. Zane. An improved exponential-time algorithm for k -SAT. *Journal of the ACM*, 52(3):337–364, May 2005.
- [PPZ97] R. Paturi, P. Pudlák, and F. Zane. Satisfiability coding lemma. In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science, FOCS 1997*, pages 566–574, 1997.
- [Pud98] P. Pudlák. Satisfiability — algorithms and logic. In *Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science, MFCS 1998*, volume 1450 of *Lecture Notes in Computer Science*, pages 129–141. Springer, 1998.
- [PY91] C. Papadimitriou and M. Yannakakis. Optimization, approximation and complexity classes. *Journal of Computer and System Sciences*, 43:425–440, 1991.
- [Rol05] D. Rolf. Derandomization of PPSZ for Unique- k -SAT. In *Proceedings of the 8th International Conference on Theory and Applications on Satisfiability Testing, SAT 2005*, volume 3569 of *Lecture Notes in Computer Science*, pages 216–225. Springer, June 2005.
- [Rol06] D. Rolf. Improved bound for the PPSZ/Schöning algorithm for 3-SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:111–122, November 2006.
- [Sch78] T. J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing, STOC 1978*, pages 216–226, 1978.
- [Sch99] U. Schöning. A probabilistic algorithm for k -SAT and constraint satisfaction problems. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science, FOCS 1999*, pages 410–414, 1999.
- [Sch02] U. Schöning. A probabilistic algorithm for k -SAT based on limited local search and restart. *Algorithmica*, 32(4):615–623, 2002.
- [Sch05] R. Schuler. An algorithm for the satisfiability problem of formulas in conjunctive normal form. *Journal of Algorithms*, 54(1):40–44, January 2005.
- [SFM78] J. I. Seiferas, M. J. Fischer, and A. R. Meyer. Separating nondeterministic time complexity classes. *Journal of the ACM*, 25(1):146–167, 1978.
- [VV86] L. Valiant and V. Vazirani. NP is as easy as detecting unique solutions. *Theoretical Computer Science*, 47:85–93, 1986.
- [Wah05] M. Wahlström. An algorithm for the SAT problem for formulae of linear length. In *Proceedings of the 13th Annual European Symposium on Algorithms, ESA 2005*, volume 3669 of *Lecture Notes in Computer Science*, pages 107–118. Springer, October 2005.

- [Wil10] R. Williams. Improving exhaustive search implies superpolynomial lower bounds. In *Proceedings of the 42nd Annual ACM Symposium on Theory of Computing, STOC 2010*, pages 231–240, 2010. Journal version: *SIAM J. Comput.*, 42(3):1218–1244, 2013.
- [Wil11] R. Williams. Non-uniform ACC circuit lower bounds. In *Proceedings of the 26th Annual IEEE Conference on Computational Complexity, CCC 2011*, pages 115–125, 2011. Journal version: *J. ACM*, 61(1):2:1–2:32, 2014.
- [Yao90] A. C. Yao. On ACC and threshold circuits. In *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science, FOCS 1990*, pages 619–627, 1990.

Chapter 17

Fixed-Parameter Tractability

Marko Samer[†] and Stefan Szeider

17.1. Introduction

The propositional satisfiability problem (SAT) is famous for being the first problem shown to be NP-complete—we cannot expect to find a polynomial-time algorithm for SAT. However, over the last two decades, SAT-solvers have become amazingly successful in solving formulas with thousands of variables that encode problems arising from various application areas. Theoretical performance guarantees, however, are far from explaining this empirically observed efficiency. Theorists believe that the trivial 2^n time bound for solving SAT instances with n variables cannot be significantly improved, say to $2^{o(n)}$ (see the end of Section 17.2.1). This enormous discrepancy between theoretical performance guarantees and the empirically observed performance of SAT solvers can be explained by the presence of a certain “hidden structure” in instances that come from applications. This hidden structure greatly facilitates the propagation and simplification mechanisms of SAT solvers. Thus, for deriving theoretical performance guarantees that are closer to the actual performance of solvers, one needs to take this hidden structure of instances into account. The literature contains several suggestions for making the vague term of a hidden structure explicit. For example, the hidden structure can be considered as the “tree-likeness” or “Horn-likeness” of the instance (below we will discuss how these notions can be made precise). All such concepts have in common that one associates with a CNF formula F a non-negative integer $k = \pi(F)$; the smaller the integer, the more structured the instance from a certain perspective. We call such a mapping π a *satisfiability parameter* or a *parameterization* of the satisfiability problem. We will also write $\pi(F) = \infty$ to express that ϕ is not defined for F . Consider a satisfiability parameter π . For each integer k one can consider the class \mathcal{C}_k^π of formulas F such that $\pi(F) \leq k$. This gives rise to an infinite hierarchy $\mathcal{C}_0^\pi \subseteq \mathcal{C}_1^\pi \subseteq \mathcal{C}_2^\pi \subseteq \dots \subseteq \mathcal{C}_\infty^\pi$ of classes. Every CNF formula F belongs to some \mathcal{C}_k^π for k sufficiently large (namely, $k = \pi(F)$).

We are interested in satisfiability parameters π such that satisfiability of instances in \mathcal{C}_k^π and membership in \mathcal{C}_k^π can be decided in polynomial time (the latter property can be relaxed). The larger we make k (thus, the more general the

[†]1977–2010

class \mathcal{C}_k^π), the worse we expect the performance guarantee for the polynomial-time algorithm for solving instances in \mathcal{C}_k^π —in other words, we expect an inevitable *tradeoff between generality and performance*.

Assume our SAT algorithm for \mathcal{C}_k^π runs in time $\mathcal{O}(n^k)$ on instances with n variables, then we have an example for a non-uniform polynomial-time algorithm, since the degree of the polynomial depends on k . On the other hand, a running time such as $\mathcal{O}(2^k n^3)$ establishes uniform polynomial time. For a non-uniform polynomial-time algorithm even relatively small values for k render classes \mathcal{C}_k^π practically infeasible—take the above example of time complexity $\mathcal{O}(n^k)$ and consider an instance $F \in \mathcal{C}_{10}^\pi$ with $n = 1000$ variables. On the other hand, a uniform polynomial-time algorithm with running time such as $\mathcal{O}(2^k n^3)$ makes the satisfiability problem practically feasible for classes \mathcal{C}_k^π as long as k remains small.

It is an intriguing research objective to design and study satisfiability parameters and to find out whether they admit uniform polynomial-time algorithms or not. Classical complexity theory does not provide the means and tools for this purpose, as the computational complexity of a problem is considered exclusively in terms of the *input size*; structural properties of instances are not represented. In the late 1980s, Rod Downey and Mike Fellows initiated the framework of *Parameterized Complexity* which resolves this shortcoming of classical theory. Their point of departure was the following observation: uniform polynomial-time algorithms exist for finding a vertex cover of size k in a graph, but apparently, no uniform polynomial-time algorithm exists for finding an independent set of size k (in both cases k is considered as the parameter). Downey, Fellows, and their collaborators have developed a rich theoretical framework for studying the computational complexity of parameterized problems. Over recent years, parameterized complexity has become an essential branch of algorithm design and analysis in both applied and theoretical areas of computer science; hundreds of research papers and several monographs have been published so far on the subject [DF99, FG06, Nie06, CFK⁺13, DF13]. Parameterized complexity considers problem instances in a *two-dimensional* setting: the first dimension is the usual input size n , the second dimension is a non-negative integer k , the *parameter*. An algorithm that solves an instance in time $\mathcal{O}(f(k)n^c)$ is called a *fixed-parameter algorithm*; here f denotes an arbitrary computable function and c denotes a constant that is independent of n and k . Thus fixed-parameter algorithms are algorithms with a uniform polynomial-time complexity as considered in the above discussion. A parameterized problem is *fixed-parameter tractable* if a fixed-parameter algorithm can solve it. Once a problem has been found to be fixed-parameter tractable, one can try to find better and better algorithms, making the function f to grow slower, and obtaining a smaller constant c . In many cases, one can observe a trajectory of steadily improved running times for parameterized problems [Fel03]. Therefore we will mainly focus on the classification of satisfiability parameters on whether or not they admit fixed-parameter tractable satisfiability decision, and not on concrete running times.

Parameterized complexity also offers a *completeness theory* which is similar to the theory of NP-completeness in the classical setting. This completeness theory provides strong evidence that certain parameterized problems (such as the parameterized independent set problem as mentioned above) are *not* fixed-parameter

tractable. We will briefly discuss some fundamental notions of this completeness theory in Section 17.2.1. In this survey, however, we will mainly focus on positive results, describing key concepts that lead to satisfiability parameters that admit fixed-parameter algorithms. The presented negative results (i.e., hardness results) have merely the purpose of carving out territories that are very likely to be inaccessible to fixed-parameter algorithms.

The majority of combinatorial problems studied in the framework of parameterized complexity offers a “natural parameter,” e.g., it is natural to parameterize the vertex cover problem by the size of the vertex cover. However, the satisfiability problem lacks a single obvious natural parameter—there are numerous possibilities for parameters. This variety, however, makes parameterized SAT a rich and exciting research area; one of its fundamental objectives is to identify satisfiability parameters that are as general as possible (i.e., for as many instances as possible one can expect that the parameter is small), and which are still accessible to fixed-parameter algorithms.

Although our main focus is satisfiability decision, we will come across several satisfiability parameters that even render the propositional model counting problem $\#\text{SAT}$ to be fixed-parameter tractable.

Next, in Section 17.2, we will provide some preliminaries: we will introduce basic notions of parameterized complexity, our terminology on CNF formulas and truth assignments, and graphs and hypergraphs associated with CNF formulas. In Section 17.3 we will introduce a general framework for parameterizing the satisfiability problem in terms satisfiability parameters, and will discuss parameterized optimization problems related to SAT. The next three sections are devoted to satisfiability parameters of different flavors: in Section 17.4 we will consider parameters based on backdoor sets relative to a polynomial-time base class; in Section 17.5 we will consider parameters that measure the “tree-likeness” of instances; in Section 17.6 we will consider further parameters including one that is based on graph matchings and one that is based on the community structure of formulas. We will conclude with final remarks in Section 17.7.

17.2. Preliminaries

17.2.1. Fixed-Parameter Algorithms

In this section, we provide a brief (and rather informal) review of some fundamental concepts of parameterized complexity. For an in-depth treatment of the subject, we refer the reader to other sources [DF99, FG06, Nie06, CFK⁺13, DF13].

An instance of a parameterized problem is a pair (I, k) where I is the *main part* and k is the *parameter*; the latter is usually a non-negative integer. A parameterized problem is *fixed-parameter tractable* if a fixed-parameter algorithm can solve it, i.e., if instances (I, k) can be solved in time $\mathcal{O}(f(k)\|I\|^c)$ where f is a computable function, c is a constant, and $\|I\|$ denotes the size of I with respect to some reasonable encoding. FPT denotes the class of all fixed-parameter tractable decision problems.

Let us illustrate the idea of a fixed-parameter algorithm through the *vertex cover* problem parameterized by the solution size. This is the best-studied prob-

lem in parameterized complexity with a long history of improvements [CKJ01]. Let us state the parameterized vertex cover problem:

VC

Instance: A graph $G = (V, E)$ and a non-negative integer k .

Parameter: k .

Task: Decide whether there is a subset $S \subseteq V$ of size at most k such that every edge of G has at least one of its incident vertices in S (S is a *vertex cover* of G).

Note that if we consider k not as a parameter but simply as a part of the input, then we get an NP-complete problem [GJ79]. A simple fixed-parameter algorithm for **VC** can be constructed as follows. Given an instance (G, k) of **VC**, we construct a binary search tree. The root of the tree is labeled with (G, k) . We choose an arbitrary edge uv of G and observe that every vertex cover of G must contain u or v . Hence we can branch into these two cases. That is, we add two children to the root, labeled with $(G - u, k - 1)$ and $(G - v, k - 1)$, respectively (k gets decremented as we have spent one unit for taking u or v into the vertex cover). We recursively extend this branching. We stop a branch of the tree if we reach a node labeled with (G', k') such that either $k' = 0$ (we have used up the budget k) or G' has no edges (we have found a vertex cover of size $k - k'$). Note that in the second case we can find the vertex cover of size $k - k'$ by collecting the vertices that have been removed from G along the path from the root to the leaf. It is easy to see the outlined algorithm is correct and decides **VC** in time $\mathcal{O}(2^k n)$ for graphs with n vertices. Using the \mathcal{O}^* -notation [Woe03] which suppresses polynomial factors, we can state the running time of the above algorithm by the expression $\mathcal{O}^*(2^k)$.

The above algorithm for **VC** illustrates the method of *bounded search trees* for the design of fixed-parameter algorithms. *Kernelization* is another essential technique, which shrinks the size of the given problem instance by employing (polynomial-time) data reduction rules until the size is bounded by a function of the parameter k . The reduced instance is called a *problem kernel*. Once a problem kernel is obtained for a decidable problem, we know that the problem is fixed-parameter tractable, since the running time of any brute force algorithm depends on the parameter k only. The converse is also true: whenever a parameterized problem is fixed-parameter tractable, then the problem admits a polynomial-time kernelization [CCDF97]. Consider the **VC** problem again as an example. It is easy to see that a vertex v of degree greater than k must belong to every vertex cover of size at most k ; hence if we have such a vertex v , we can reduce the instance (G, k) to $(G - v, k - 1)$. Assume that we are left with the instance (G', k') after we have applied the reduction rule as long as possible (if $k' < 0$, then we reject the instance). Observe that each vertex of G' can cover at most k edges. Hence, if G' has more than k^2 edges, we know that G has no vertex cover of size at most k . On the other hand, if G' has at most k^2 edges, we have a problem kernel that can be solved by brute force.

The current best worst-case time complexity for **VC** is due to Chen, Kanj, and Xia [CKX10]. The algorithm is based on more sophisticated kernelization rules and achieves a running time of $\mathcal{O}^*(1.2738^k)$. Further information on ker-

nelization can be found in Guo and Niedermeier's survey [GN07]. Gaspers and Szeider [GS14] gave several kernelization results that are specifically relevant for parameterized satisfiability.

Next, we turn our attention to fixed-parameter *intractability*, to problems that are believed to be not fixed-parameter tractable. Consider for example the following parameterized independent set problem.

IS

Instance: A graph $G = (VE)$ and a non-negative integer k .

Parameter: k .

Task: Decide whether there is a subset $S \subseteq V$ of size at least k such that no edge of G joins two vertices in S (S is an *independent set* of G).

No fixed-parameter algorithm for this problem is known, and there is strong evidence to believe that no such algorithm exists [DF99]. For example, fixed-parameter tractability of **IS** would imply the existence of an $\mathcal{O}^*(2^{o(n)})$ -time algorithm for the n -variable 3-SAT problem [FG06]. The assumption that the latter is not the case is known as the *Exponential Time Hypothesis (ETH)* [IPZ01]; see also Chapter 16 for more information on the ETH.

Thus, **VC** is fixed-parameter tractable, whereas **IS** is believed to be not. Note, however, that under classical polynomial-time many-to-one reductions, **VC** and **IS** are equivalent for trivial reasons: a graph with n vertices has a vertex cover of size k if and only if it has an independent set of size $k' = n - k$. Hence, to distinguish between fixed-parameter tractable and fixed-parameter intractable problems, one needs a notion of reduction that restricts the way of how parameters are mapped to each other. An *fpt-reduction* from a parameterized decision problem L to a parameterized decision problem L' is an algorithm that transforms an instance (I, k) of L into an instance $(I', g(k))$ of L' in time $\mathcal{O}(f(k) \|I\|^c)$ (f, g are arbitrary computable functions, c is an arbitrary constant), such that (I, k) is a yes-instance of L if and only if $(I', g(k))$ is a yes-instance of L' . It is easy to see that indeed, if L' is fixed-parameter tractable and there is an fpt-reduction from L to L' , then L is fixed-parameter tractable as well. Note that the reduction from **VC** to **IS** as sketched above is not an fpt-reduction since $k' = n - k$ and so k' is not a function of k alone.

The class of problems that can be reduced to **IS** under fpt-reductions is denoted by $W[1]$. A problem is called *W[1]-hard* if **IS** (and so every problem in $W[1]$) can be reduced to it by an fpt-reduction. A problem is called *W[1]-complete* if it is $W[1]$ -hard and belongs to $W[1]$. Thus, a problem is $W[1]$ -complete if and only if it is equivalent to **IS** under fpt-reductions. A similar terminology applies to other parameterized complexity classes.

Consider the following parameterized hitting set problem (it is the basis for several hardness results that we will consider in the remainder of this chapter).

HS

Instance: A family \mathcal{S} of finite sets S_1, \dots, S_m and a non-negative integer k .

Parameter: k .

Task: Decide if there is a subset $R \subseteq \bigcup_{i=1}^m S_i$ of size at most k such that $R \cap S_i \neq \emptyset$ for all $i = 1, \dots, m$ (R is a *hitting set* of \mathcal{S}).

Observe that, indeed, a search tree algorithm as outlined above for **VC** does not yield fixed-parameter tractability for **HS**: since the size of the sets S_i is unbounded, a search tree algorithm would entail an unbounded branching factor. If, however, the size of the sets S_i is bounded by some constant q , then the problem (known as q -**HS**) becomes fixed-parameter tractable. The obvious search tree algorithm has time complexity $\mathcal{O}^*(q^k)$. For $q = 3$, Wahlström [Wah17] developed a fixed-parameter algorithm with running time $\mathcal{O}^*(2.076^k)$.

HS is $W[1]$ -hard, but no fpt-reduction from **HS** to **IS** is known, and it is believed that such a reduction does not exist. In other words, **HS** appears to be harder than the problems in $W[1]$. The class of problems that can be reduced to **HS** under fpt-reductions is denoted by $W[2]$. In fact, $W[1]$ and $W[2]$ form the first two levels of an infinite chain of classes $W[1] \subseteq W[2] \subseteq W[3] \subseteq \dots \subseteq W[P]$, the so-called “weft hierarchy.” All inclusions are believed to be proper. There are several sources of theoretical evidence for assuming that the classes of the weft hierarchy are distinct from FPT: accumulating evidence [Ces06], evidence based on parameterized analogs of Cook’s Theorem [DF99], and evidence obtained by proof complexity methods [DMS11].

We say that a parameterized problem is *XP-tractable* if instances (I, k) can be solved in time $\mathcal{O}(\|I\|^{f(k)})$ for a computable function f . The class XP consists of all XP-tractable parameterized decision problems. $FPT \neq XP$ is provably true [DF99, FG06]. Together with the classes of the weft hierarchy, we have the following chain of inclusions:

$$FPT \subseteq W[1] \subseteq W[2] \subseteq \dots \subseteq W[P] \subseteq XP.$$

17.2.2. CNF Formulas and Truth Assignments

Before discussing parameterizations of SAT, let us introduce some notation and basic notions related to SAT. We consider propositional formulas in conjunctive normal form (CNF), short *CNF formulas* or just *formulas*, represented as a finite set of *clauses*. A clause is a finite set of *literals*, and a literal is a negated or un-negated propositional *variable*. For a literal ℓ we denote by $\bar{\ell}$ the literal of opposite polarity, i.e., $\bar{x} = \neg x$ and $\overline{\neg x} = x$. We also write $x^1 = x$ and $x^0 = \neg x$. Similarly, for a set L of literals, we put $\bar{L} = \{\bar{\ell} : \ell \in L\}$. We say that two clauses C, D *overlap* if $C \cap D \neq \emptyset$, and we say that C and D *clash* if C and \bar{D} overlap. For a clause C we denote by $\text{var}(C)$ the set of variables that occur (negated or un-negated) in C ; for a formula F we put $\text{var}(F) = \bigcup_{C \in F} \text{var}(C)$. We measure the *size* $\|F\|$ of a formula F by its *length* $\sum_{C \in F} |C|$.

CNF formulas F and F' are *isomorphic* if they differ only in the name of variables. That is, if $F = \{C_1, \dots, C_m\}$, $F' = \{C'_1, \dots, C'_m\}$, and there is a one-to-one mapping $f : \text{var}(F) \rightarrow \text{var}(F')$ such that $C'_i = \{(f(x))^\varepsilon : x^\varepsilon \in C_i, x \in \text{var}(F), \varepsilon \in \{0, 1\}\}$ holds for all $1 \leq i \leq m$.

CNF formulas F and F' are *renamings* of each other if there exists a set $X \subseteq \text{var}(F)$ such that F' can be obtained from F by flipping the polarity of all literals $\ell \in X \cup \bar{X}$. That is, if $F = \{C_1, \dots, C_m\}$, $F' = \{C'_1, \dots, C'_m\}$, and $C'_i = \{\ell : \ell \in C_i \setminus (X \cup \bar{X})\} \cup \{\bar{\ell} : \ell \in C_i \cap (X \cup \bar{X})\}$.

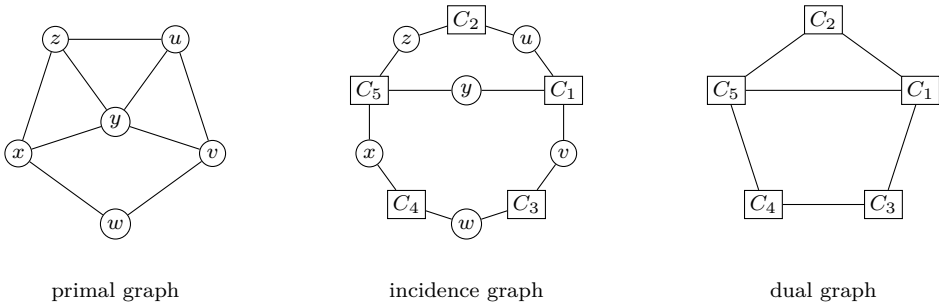


Figure 17.1. Graphs associated with the CNF formula $F = \{C_1, \dots, C_5\}$ with $C_1 = \{u, \neg v, \neg y\}$, $C_2 = \{\neg u, z\}$, $C_3 = \{v, \neg w\}$, $C_4 = \{w, \neg x\}$, $C_5 = \{x, y, \neg z\}$; the primal graph $G(F)$, the incidence graph $G^*(F)$, and the dual graph $G^d(F)$.

A *truth assignment* is a mapping $\tau : X \rightarrow \{0, 1\}$ defined on some set X of variables. If $X = \{x\}$ we denote τ simply by “ $x = 1$ ” or “ $x = 0$ ”. We extend τ to literals by setting $\tau(\neg x) = 1 - \tau(x)$ for $x \in X$. $F[\tau]$ denotes the formula obtained from F by removing all clauses that contain a literal ℓ with $\tau(\ell) = 1$ and by removing from the remaining clauses all literals ℓ' with $\tau(\ell') = 0$. $F[\tau]$ is the *restriction* of F to τ . Note that $\text{var}(F[\tau]) \cap X = \emptyset$ holds for every truth assignment $\tau : X \rightarrow \{0, 1\}$ and every formula F . A truth assignment $\tau : X \rightarrow \{0, 1\}$ *satisfies* a clause C if $\tau(\ell) = 1$ for at least one literal $\ell \in C$; τ satisfies a formula F if it satisfies all clauses of F . Note that τ satisfies F if and only if $F[\tau] = \emptyset$. A formula F is *satisfiable* if there exists a truth assignment that satisfies F ; otherwise, F is *unsatisfiable*. A truth assignment $\tau : \text{var}(F) \rightarrow \{0, 1\}$ is called *total* for the formula F . A satisfying total truth assignment of F is called a *model* of F . We denote the number of models of a formula F by $\#(F)$. Two formulas are *equisatisfiable* if either both are satisfiable or both are unsatisfiable. **SAT** is the problem of deciding whether a given formula is satisfiable. **\#SAT** is the problem of determining the number of models of a given formula.

Let $x \in \text{var}(F)$ and $\varepsilon \in \{0, 1\}$. If $\{x^\varepsilon\} \in F$, then F and $F[x = \varepsilon]$ are equisatisfiable; $F[x = \varepsilon]$ is said to be obtained from F by *unit propagation*. If some clause of F contains x^ε but none contains $x^{1-\varepsilon}$, then x^ε is called a *pure literal* of F . If x^ε is a pure literal of F , then obviously F and $F[x = \varepsilon]$ are equisatisfiable. In that case, we say that $F[x = \varepsilon]$ is obtained from F by *pure literal elimination*.

17.2.3. Graphs and Hypergraphs Associated with CNF Formulas

In this section, we will discuss several graphs and hypergraphs that can be used to represent the structure of a CNF formula.

Perhaps the most prominent graph representation of a CNF formula F is the *primal graph* $G(F)$. The vertices of $G(F)$ are the variables of F ; two variables x, y are joined by an edge if they occur in the same clause, that is, if $x, y \in \text{var}(C)$ for some $C \in F$. Another important graph is the *incidence graph* $G^*(F)$. The vertices of $G^*(F)$ are the variables and clauses of F ; a variable x and a clause C

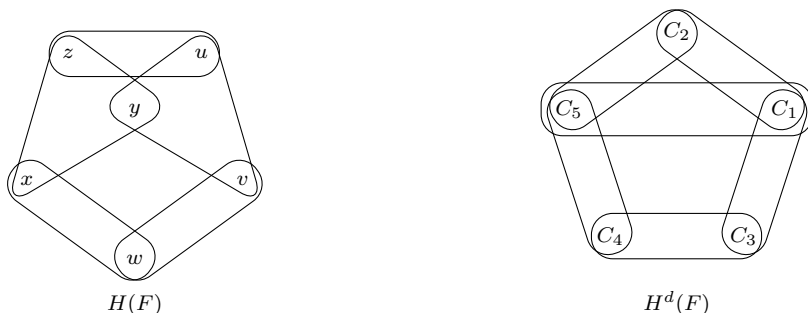


Figure 17.2. The hypergraph $H(F)$ and the dual hypergraph $H^d(F)$ associated with the CNF formula F of Figure 17.1.

are joined by an edge if $x \in \text{var}(C)$. If we orient the edges of the incidence graph according to the polarity of the literals, then we obtain the *directed incidence graph* $D^*(F)$: an edge is directed from C to x if $x \in C$, and directed from x to C if $\neg x \in C$. In analogy to the primal graph, one can also define the *dual graph* $G^d(F)$. The vertices of $G^d(F)$ are the clauses of F ; two clauses C_1, C_2 are joined by an edge if there is a variable occurring in both of them, that is, if $x \in \text{var}(C_1) \cap \text{var}(C_2)$ for some $x \in \text{var}(F)$. We also consider the following two graphs, the *conflict graph* $G^{\ddagger}(F)$ and the *consensus graph* $G^{**}(F)$, both having as vertices the clauses of F , where in $G^{\ddagger}(F)$ two clauses C_1, C_2 are joined by an edge if they clash ($C_1 \cap C_2 \neq \emptyset$), and in $G^{**}(F)$ two clauses C_1, C_2 are joined by an edge if they do not clash ($C_1 \cap \overline{C_2} = \emptyset$).

Figure 17.1 shows the primal graph, the incidence graph, and the dual graph of a CNF formula.

Hypergraphs generalize graphs in the sense that each edge may connect more than just two vertices, i.e., the edges (called *hyperedges*) of a hypergraph are non-empty sets of vertices. We associate to each CNF formula F its *hypergraph* $H(F)$. The vertices of $H(F)$ are the variables of F and for each $C \in F$ the set $\text{var}(C)$ represents a hyperedge of $H(F)$. The *dual hypergraph* $H^d(F)$ is defined symmetrically: the vertices of $H^d(F)$ are the clauses of F ; for each variable x , the set of clauses C with $x \in \text{var}(C)$ forms a hyperedge. See Figure 17.2 for examples.

17.3. Parameterized SAT

In the first part of this section we will develop the framework for parameterized SAT decision and model counting, where the parameter represents structural information of the instances. This framework will be used throughout the remainder of this chapter. In the second part of this section we will discuss parameterized optimization problems that are related to satisfiability.

17.3.1. Satisfiability Parameters

A *satisfiability parameter* is a computable function π that assigns to every formula F a non-negative integer $\pi(F)$. We assume that $\pi(F) = \pi(F')$ if two formulas F, F' are isomorphic (see Section 17.2.2), i.e., π is invariant with respect to isomorphisms.

We are interested in satisfiability parameters that allow fixed-parameter tractability of satisfiability decision for instances F with respect to the parameter $k = \pi(F)$. Accordingly, for a satisfiability parameter π we consider the following generic parameterized problem.

SAT(π)

Instance: A formula F and a non-negative integer k .

Parameter: k .

Task: Decide whether F is satisfiable, or determine that $\pi(F) > k$.

This way of parameterizing the satisfiability problem was suggested by Szeider [Sze04b]. It avoids the requirement of exactly recognizing formulas F with $\pi(F) \leq k$ and is related to the concept of a “robust algorithm” introduced by Spinrad [Spi03]. Marx and Schlotter [MS10, MS11] used the predicate “*permissive*” to describe parameterized problems of this form, in contrast to “*strict*” problems, which entail the exact determination of the parameter value. We can, as well, formulate model counting as a permissive problem:

#SAT(π)

Instance: A formula F and a non-negative integer k .

Parameter: k .

Task: Compute $\#(F)$ or determine that $\pi(F) > k$.

If we solve a parameterized satisfiability problem in the strict sense, we have to solve the parameterized *verification problem*:

VER(π)

Instance: A formula F and a non-negative integer k .

Parameter: k .

Task: Decide whether $\pi(F) \leq k$.

We assume that an affirmative solution to the verification problem is backed up with a certain *witness* for “ $\pi(F) \leq k$ ” in form of an auxiliary structure, which can subsequently be used for deciding the satisfiability of F or counting the models of F .

In some cases, **VER**(π) is W[1]-hard or just not known to be fixed-parameter tractable, but we can still use π for fixed-parameter tractability of **SAT**(π) or **#SAT**(π) if **VER**(π) is *fixed-parameter approximable*. That is, if there is a computable function g and a fixed-parameter algorithm that, given F and k , either outputs a witness for $\pi(F) \leq g(k)$, or correctly decides that $\pi(F) > k$. We can use the witness for $\pi(F) \leq g(k)$ as an input for a parameterized algorithm that decides the satisfiability of F or computes $\#(F)$, albeit the running time is worse than if we had a witness for $\pi(F) \leq k$.

Usually it is very difficult to establish parameterized hardness results for the permissive problems **SAT**(π) and **#SAT**(π), or to provide theoretical evidence

that $\mathbf{VER}(\pi)$ is not fixed-parameter approximable. We have only very few examples for such hardness results (see, e.g., Theorems 17.4.5, 17.4.8, and 17.4.5). Although a parameterized hardness result for $\mathbf{VER}(\pi)$ does not rule out the fixed-parameter tractability of $\mathbf{SAT}(\pi)$, it still provides the valuable insight that the fixed-parameter tractability of $\mathbf{SAT}(\pi)$ in the strict sense is unlikely.

The notion of *dominance* allows us to compare two satisfiability parameters π and π' with respect to their generality. We say that π *dominates* π' if there exists a computable function f such that for every formula F we have

$$\pi(F) \leq f(\pi'(F)).$$

Furthermore, π *strictly dominates* π' if π dominates π' but not vice versa. Finally, π and π' are *domination incomparable* if neither dominates the other, and *domination equivalent* if they dominate each other. If π strictly dominates π' we also say that π is *more general* than π' . Dominance, strict dominance, and domination equivalence are transitive relations between satisfiability parameters. Furthermore, since strict dominance is antisymmetric, it gives rise to a partial ordering of satisfiability parameters. The next result follows directly from the definitions.

Lemma 17.3.1. *If π dominates π' , then there is an fpt-reduction from $\mathbf{SAT}(\pi')$ to $\mathbf{SAT}(\pi)$.*

Thus, if $\mathbf{SAT}(\pi)$ is fixed-parameter tractable and π dominates π' , then $\mathbf{SAT}(\pi')$ is also fixed-parameter tractable. It is an important goal to find satisfiability parameters π that are as general as possible and for which both problems $\mathbf{SAT}(\pi)$ and $\mathbf{VER}(\pi)$ are fixed-parameter tractable. We conclude this section with three trivial examples of satisfiability parameters.

Example 17.3.1. Let $\mathbf{n}(F)$ denote the number of variables of a formula F . The verification problem $\mathbf{VER}(\mathbf{n})$ is trivial. The obvious algorithm that considers all possible truth assignments of F runs in time $\mathcal{O}^*(2^{\mathbf{n}(F)})$ and is, therefore, a fixed-parameter algorithm with respect to \mathbf{n} . Hence $\mathbf{SAT}(\mathbf{n})$ is fixed-parameter tractable. \square

A satisfiability parameter π becomes an interesting one if every class \mathcal{C}_k^π contains formulas with an arbitrarily large number of variables; i.e., if π is more general than the satisfiability parameter \mathbf{n} considered in the example above.

Example 17.3.2. Let $\mathbf{ml}(F)$ denote the *maximum length* of clauses in a SAT instance F (with $\mathbf{ml}(F) = 0$ if $F = \emptyset$). From the NP-completeness of 3SAT it follows that $\mathbf{SAT}(\mathbf{ml})$ is not fixed-parameter tractable unless $\mathbf{P} = \mathbf{NP}$. So $\mathbf{SAT}(\mathbf{ml})$ is probably not even in XP. \square

Example 17.3.3. Let \mathcal{A} be a deterministic polynomial-time algorithm that applies polynomial-time simplification and propagation rules to a formula without changing its satisfiability. Say, the algorithm applies *unit propagation* and *pure literal elimination* as long as possible (see Section 17.2.2 above), plus possibly some further rules. For more powerful preprocessing rules see, e.g., the work of Bacchus and Winter [BW04]. Let $\mathcal{A}(I)$ denote the instance obtained from I by

applying algorithm \mathcal{A} , and let $\mathbf{n}_{\mathcal{A}}(I)$ denote the number of variables of $\mathcal{A}(I)$ (if $\mathcal{A}(I)$ depends on a particular ordering of variables and clauses, let $\mathbf{n}_{\mathcal{A}}(I)$ denote the largest number over all such orderings).

It is easy to see that the problem $\mathbf{SAT}(\mathbf{n}_{\mathcal{A}})$ is fixed-parameter tractable since after the polynomial-time preprocessing we are left with an instance $\mathcal{A}(I)$ whose number of variables is bounded in terms of the parameter k , and therefore any brute-force algorithm applied to $\mathcal{A}(I)$ is a fixed-parameter algorithm. In other words, $\mathbf{SAT}(\mathbf{n}_{\mathcal{A}})$ is fixed-parameter tractable since the preprocessing provides a kernelization. $\mathbf{VER}(\mathbf{n}_{\mathcal{A}})$ is easy, as we just need to count the number of variables left after applying the polynomial-time preprocessing algorithm \mathcal{A} . Clearly $\mathbf{n}_{\mathcal{A}}$ is more general than \mathbf{n} (Example 17.3.1) since one can easily find formulas where, say, unit propagation eliminates an arbitrarily large number of variables. \square

17.3.2. Optimization Problems

Max-SAT is the optimization version of the satisfiability problem, where, given a CNF formula F and an integer k , one asks whether there is a truth assignment that satisfies at least k clauses of F . In the classical setting, Max-SAT is NP-complete even if all clauses contain at most two literals (Max-2-SAT). What happens if we consider k as the parameter?

Under this parameterization, Max-SAT is easily seen to be fixed-parameter tractable (we roughly follow [MR99]). Let (F, k) be an instance of Max-SAT. For a truth assignment τ we write $s(\tau)$ for the number of clauses of F that are satisfied by τ . Moreover, let τ^* denote the extension of τ that includes all variable assignments obtained by (iterated and exhaustive) application of pure literal elimination. For example, if $F = \{\{w, \bar{x}\}, \{x, y, \bar{z}\}, \{\bar{y}, z\}, \{\bar{w}, \bar{z}\}\}$ and $\tau = \{(w, 1)\}$, then $\tau^* = \{(w, 1), (x, 1), (y, 0), (z, 0)\}$. We construct a binary search tree T whose nodes are truth assignments. We start with the empty assignment as the root and extend the tree downwards as follows. Consider a node τ of T . If $s(\tau^*) \geq k$ or if $s(\tau^*) < k$ and $F[\tau^*] = \{\emptyset\}$, then we do not add any children to τ ; in the first case we label τ as “success leaf,” in the second case as “failure leaf.” Otherwise, if $s(\tau^*) < k$ and $F[\tau^*] \neq \{\emptyset\}$, we pick a variable $x \in F[\tau^*]$ and we add below τ the children $\tau_0 = \tau \cup \{(x, 0)\}$ and $\tau_1 = \tau \cup \{(x, 1)\}$. Note that in this case both $s(\tau_0)$ and $s(\tau_1)$ are strictly greater than $s(\tau)$. It is easy to see that there exists a total truth assignment τ of F that satisfies at least k clauses if and only if T has a success leaf (for the only-if direction note that τ defines a path from the root of T to a success leaf). Since at each branching step the number of satisfied clauses increases, it follows that T is of depth at most k and so has at most 2^k leaves. Hence the search algorithm runs in time $\mathcal{O}^*(2^k)$ which renders Max-SAT fixed-parameter tractable for parameter k . By sophisticated case distinctions, one can make the algorithm significantly faster. The currently fastest algorithm is runs in time $\mathcal{O}^*(1.3248^k)$ [CXW17].

Note that one can always satisfy at least half of the clauses of a CNF formula (the all-true or the all-false assignment will do). Thus, a more challenging parameter for Max-SAT is the number $k - |F|/2$ (this constitutes a parameterization “above the guaranteed value” $|F|/2$). Indeed, by a result of Mahajan and Raman [MR99], Max-SAT is fixed-parameter tractable also under this more general setting. For an r -CNF input formula, where r is an arbitrary constant, one

can always satisfy at least a $1 - 2^{-r}$ fraction of its clauses, and such an assignment can be found in polynomial time using Johnson's Algorithm [Joh73]. Alon et al. [AGK⁺11] showed that Max-SAT for r -CNF formulas is fixed-parameter tractable parameterized above this bound, answering a question posed by Mahajan et al. [MRS06].

One can consider an even more challenging approach, taking the *dual parameter* $k' = |F| - k$; that is, to parameterize by the number of clauses that remain unsatisfied. It is easy to see that for every *constant* k' the problem is NP-complete in general and polynomial-time solvable for 2CNF formulas (Max-2-SAT). It follows from recent results of Razgon and O'Sullivan [RO08] that Max-2-SAT is fixed-parameter tractable for the dual parameter k' . We will return to this problem again in Section 17.4.1.

Apart from Max-SAT there are also other interesting optimization versions of satisfiability. For example, *Bounded-CNF-SAT* asks whether a CNF formula can be satisfied by setting at most k variables to true. With parameter k , Bounded-CNF-SAT is W[2]-complete; Bounded-3-CNF-SAT, however, is easily seen to be fixed-parameter tractable [DMS11]. A similar problem, *Weighted-CNF-SAT*, asks for a satisfying assignment that sets *exactly* k variables to true. Weighted-CNF-SAT is W[2]-complete; Weighted- c -CNF-SAT is W[1]-complete for every constant $c \geq 2$ [DF99].

17.4. Backdoor Sets

As outlined in the introduction, every satisfiability parameter π gives rise to the hierarchy of classes

$$\mathcal{C}_0^\pi \subseteq \mathcal{C}_1^\pi \subseteq \mathcal{C}_2^\pi \subseteq \dots$$

where class \mathcal{C}_k^π contains all CNF formulas F with $\pi(F) \leq k$. We call this hierarchy the π -*hierarchy*, and we refer to the class at the lowest level of the hierarchy as the *base class*. The following are necessary conditions for a class \mathcal{C} of CNF formulas under which it could possibly act as the base class for the π -hierarchy of some satisfiability parameter π such that both **SAT**(π) and **VER**(π) are fixed-parameter tractable:

1. \mathcal{C} is closed under isomorphism;
2. membership in \mathcal{C} can be decided in polynomial time;
3. satisfiability of elements of \mathcal{C} can be decided in polynomial time.

Some authors also require that a base class is *self-reducible*, that is, if $F \in \mathcal{C}$ then $F[x = 0], F[x = 1] \in \mathcal{C}$ for all $x \in \text{var}(F)$. Most natural base classes are self-reducible.

Next, we will see how one can define a π -hierarchy starting at an arbitrary base class \mathcal{C} utilizing the notion of “backdoor sets” which was introduced by Williams, Gomes, and Selman [WGS03] for analyzing the behavior of SAT algorithms. Actually, with different terminology and context, backdoor sets have already been studied by Crama, Elkin, and Hammer [CEH97]. Consider a CNF formula F and a set $B \subseteq \text{var}(F)$ of variables. B is called a *strong \mathcal{C} -backdoor set* of F if for every truth assignment $\tau : B \rightarrow \{0, 1\}$ the restriction $F[\tau]$ belongs

to the base class \mathcal{C} . We also introduce the notion of *weak* backdoor sets. A set $B \subseteq \text{var}(F)$ is called a *weak \mathcal{C} -backdoor set* of F if there exists truth assignment $\tau : B \rightarrow \{0, 1\}$ such that the restriction $F[\tau]$ is *satisfiable* and belongs to the base class \mathcal{C} . We denote the size of a smallest strong \mathcal{C} -backdoor set of F by $\mathbf{b}_{\mathcal{C}}(F)$ and the size of a smallest weak \mathcal{C} -backdoor set of F by $\mathbf{wb}_{\mathcal{C}}(F)$.

Example 17.4.1. Consider the base class HORN of Horn formulas (an instance is Horn if each of its clauses contains at most one un-negated variable) and consider the formula $F = \{\{u, v, w\}, \{\bar{u}, x, \bar{y}\}, \{u, \bar{v}, \bar{x}, y\}, \{v, y, \bar{z}\}, \{u, v, \bar{w}, z\}\}$. The set $B = \{u, v\}$, is a strong HORN-backdoor set since $F[\tau] \in \text{HORN}$ for all four truth assignments $\tau : B \rightarrow \{0, 1\}$. \square

Note that F is satisfiable if and only if at least one of the restrictions $F[\tau]$, $\tau : B \rightarrow \{0, 1\}$, is satisfiable. Thus, if we know a strong \mathcal{C} -backdoor set B of F , we can decide the satisfiability of F by deciding the satisfiability of at most $2^{|B|}$ polynomial-time solvable formulas that belong to \mathcal{C} (this is a $\mathcal{O}^*(2^k)$ fixed-parameter algorithm with respect to the parameter $k = |B|$). Of course we can find a \mathcal{C} -backdoor set of size at most k (or decide that it does not exist) by trying all subsets $B \subseteq \text{var}(F)$ with $|B| \leq k$, and checking whether all $F[\tau]$, $\tau : B \rightarrow \{0, 1\}$, belong to \mathcal{C} ; consequently $\mathbf{VER}(\mathbf{b}_{\mathcal{C}}) \in \text{XP}$. However, as we shall see in the following section, $\mathbf{VER}(\mathbf{b}_{\mathcal{C}})$ can or cannot be fixed-parameter tractable, depending on the base class \mathcal{C} .

For an algorithm that provides fixed-parameter tractability of $\mathbf{VER}(\mathbf{b}_{\mathcal{C}})$, we will always assume that it provides, as a witness for $\mathbf{b}_{\mathcal{C}}(C) \leq k$, a strong \mathcal{C} -backdoor set of F size of size $\leq k$. Hence the fixed-parameter tractability of $\mathbf{VER}(\mathbf{b}_{\mathcal{C}})$ implies the fixed-parameter tractability of $\mathbf{SAT}(\mathbf{b}_{\mathcal{C}})$.

As mentioned above, a strong \mathcal{C} -backdoor set of F of size k reduces the satisfiability of F to the satisfiability of at most 2^k instances in \mathcal{C} . The notions of *backdoor trees* and *backdoor DNFs* [SS08, OSS21] make this reduction explicit. This allows a refined worst-case estimation of the number of instances in \mathcal{C} that need to be checked, which can be exponentially smaller than 2^k .

17.4.1. Horn, 2CNF, and Generalizations

HORN and 2CNF are two important base classes for which the detection of strong backdoor sets is fixed-parameter tractable.

Theorem 17.4.1 ([NRS04]). *For $\mathcal{C} \in \{\text{HORN}, 2\text{CNF}\}$ the problems $\mathbf{SAT}(\mathbf{b}_{\mathcal{C}})$ and $\mathbf{VER}(\mathbf{b}_{\mathcal{C}})$ are fixed-parameter tractable.*

The algorithms of Nishimura et al. rely on the concept of *variable deletion*. For explaining this, it is convenient to consider the following variant of backdoor sets: A set $B \subseteq \text{var}(F)$ is called a *deletion \mathcal{C} -backdoor set* of F if $F - B$ belongs to \mathcal{C} . Here $F - B$ denotes the CNF formula $\{C \setminus (B \cup \bar{B}) : C \in F\}$, i.e., the formula obtained from F by removing from the clauses all literals of the form ℓ or $\bar{\ell}$ for $\ell \in B$. Let $\mathbf{db}_{\mathcal{C}}(F)$ denote the size of a smallest *deletion \mathcal{C} -backdoor set* of F . For many important base classes \mathcal{C} , deletion \mathcal{C} -backdoor sets are also strong \mathcal{C} -backdoor sets. In particular, this is the case if the base class is *clause induced*, i.e., if whenever F belongs to \mathcal{C} , all subsets of F belong to \mathcal{C} as well. If \mathcal{C} is clause induced, then $F[\tau] \subseteq F - B$ holds for every $\tau : B \rightarrow \{0, 1\}$.

Lemma 17.4.2. *Let \mathcal{C} be a clause-induced base class and let F be an arbitrary formula. Then every deletion \mathcal{C} -backdoor set of F is also a strong \mathcal{C} -backdoor set of F .*

For example, the base classes HORN and 2CNF are clause induced. For these two base classes, even the converse direction of Lemma 17.4.2 holds.

Lemma 17.4.3 ([CEH97, NRS04]). *Let $\mathcal{C} \in \{\text{HORN}, 2\text{CNF}\}$ and let F be an arbitrary formula. Then the strong \mathcal{C} -backdoor sets of F are exactly the deletion \mathcal{C} -backdoor sets of F .*

Example 17.4.2. Consider the formula F of Example 17.4.1 and the strong HORN-backdoor set $B = \{u, v\}$ of F (note that B is also a strong 2CNF-backdoor set of F). Indeed, $F - B = \{\{w\}, \{x, \bar{y}\}, \{\bar{x}, y\}, \{y, \bar{z}\}, \{\bar{w}, z\}\}$ is a Horn formula. \square

Nishimura et al. describe a fixed-parameter algorithm for the detection of strong HORN-backdoor sets. Their algorithm is based on bounded search trees similarly to the vertex cover algorithm described above. In fact, we can directly use a vertex cover algorithm. To this end, we associate with a formula F the *positive primal graph* G . The vertices of G are the variables of F , and two variables x, y are joined by an edge if and only if $x, y \in C$ for some clause C of F (negative occurrences of variables are ignored). Clearly, the positive primal graph can be constructed in time polynomial in the size of F . Now it is easy to see that for sets $B \subseteq \text{var}(F)$ the following properties are equivalent:

1. B is a strong HORN-backdoor set of F ;
2. B is a deletion HORN-backdoor set of F ;
3. B is a vertex cover of G .

Thus, any vertex cover algorithm, such as the $\mathcal{O}^*(1.2738^k)$ algorithm by Chen et al. [CKX10] mentioned above, can be used to find a strong HORN-backdoor set.

For the detection of strong 2CNF-backdoor sets one can apply a similar approach. Given a CNF formula F and a positive integer k , we want to determine whether F has a strong 2CNF-backdoor set of size at most k . Let \mathcal{S} be the set of all size-3 subsets S of $\text{var}(F)$ such that $S \subseteq \text{var}(C)$ for some clause C of F . Evidently, \mathcal{S} can be constructed in polynomial time. Observe that a set $B \subseteq \text{var}(F)$ is a hitting set of \mathcal{S} if and only if B is a deletion 2CNF-backdoor set of F . By Lemma 17.4.3, the latter is the case if and only if B is a strong 2CNF-backdoor set of F . Thus, Wahlström's algorithm for **3-HS** [Wah17] solves **VER**($\mathbf{b}_{2\text{CNF}}$) in time $\mathcal{O}^*(2.076^k)$.

A generalization of backdoor sets, in particular HORN- and 2CNF-backdoor sets, to quantified Boolean formulas has been proposed by taking the variable dependencies caused by the quantifications into account [SS09].

The classes HORN and 2CNF are two of the five classes of CNF formulas that can be identified with tractable satisfiability problems considered by Schaefer in his seminal work on generalized satisfiability problems [Sch78]. The remaining three are the classes HORN^- of *anti-Horn* formulas (each clause contains at most one negative literal), and for $\varepsilon \in \{0, 1\}$ the classes $\varepsilon\text{-VAL}$ of ε -*valid* formulas. A CNF-formula F is ε -*valid* if each nonempty clause $\emptyset \neq C \in F$ contains at

least one literal x^ε with $x \in \text{var}(C)$. The affine Boolean formulas considered by Schaefer do not correspond naturally to a class of CNF formulas; hence we do not consider them here.

For a given formula F we can compute in polynomial time a smallest deletion ε -VAL-backdoor set (which is also a smallest strong ε -VAL-backdoor set) by taking the union of $\text{var}(C)$ over all $C \in F$ with $C \subseteq \{x^{1-\varepsilon} : x \in \text{var}(C)\}$. We put

$$\text{Schaefer} = \{\text{HORN}, \text{HORN}^-, \text{2CNF}, \text{0-VAL}, \text{1-VAL}\}$$

and summarize the discussed results as follows.

Theorem 17.4.4. *The problem $\text{SAT}(\mathbf{b}_C)$ is fixed-parameter tractable for all $C \in \text{Schaefer}$.*

The detection of weak \mathcal{C} -backdoor sets, however, is $W[2]$ hard for several base classes, including the classes in Schaefer . Gaspers and Szeider gave a generic reduction [GS12b, Proposition 1] that can be instantiated for various base classes \mathcal{C} . For the classes $C \in \text{Schaefer}$ hardness results even for the permissive problems $\text{SAT}(\mathbf{wb}_C)$ are known:

Theorem 17.4.5 ([GS12b]). *$\text{SAT}(\mathbf{wb}_C)$ is $W[1]$ -hard for all $C \in \text{Schaefer}$*

A significant improvement over HORN as the base class for strong backdoor sets is the consideration of the class UP of CNF formulas that can be decided by unit propagation. That is, a CNF formula F belongs to UP if and only if after repeated application of unit propagation one is either left with the empty formula (i.e., F is satisfiable) or with a formula that contains the empty clause (i.e., F is unsatisfiable). Unfortunately, $\mathbf{VER}(\mathbf{b}_{\text{UP}})$ turns out to be complete for the class $W[P]$. This holds also true if one considers the base class PL of CNF formulas decidable by pure literal elimination, and by the base class $\text{UP} + \text{PL}$ of CNF formulas decidable by a combination of unit propagation and pure literal elimination. Thus $\text{UP} + \text{PL}$ contains exactly those formulas that can be decided by the polynomial-time “subsolver” of the basic DPLL procedure [WGS03].

The following result provides strong evidence that the detection of strong backdoor sets with respect to the base classes PL , UP , and $\text{UP} + \text{PL}$ is not fixed-parameter tractable. Let us write

$$\text{Subsolver} = \{\text{PL}, \text{UP}, \text{UP} + \text{PL}\}.$$

Theorem 17.4.6 ([Sze05]). *For $C \in \text{Subsolver}$, the problem $\mathbf{VER}(\mathbf{b}_C)$ is $W[P]$ -complete.*

Given this result, it appears to be very unlikely that one can find a size- k strong backdoor set with respect to the base class of formulas decidable by DPLL subsolvers significantly faster than by trying all sets of variables of size k . Also, the consideration of deletion backdoor sets does not offer an opportunity for overcoming this limitation: the classes UP , PL , and $\text{UP} + \text{PL}$ are not clause induced—indeed, not every deletion backdoor set is a strong backdoor set with respect to these classes.

However, the class RHORN of *renamable Horn* formulas is an interesting base class that is clause induced. A formula is renamable Horn if some renaming

of it is Horn. It is well known that recognition and satisfiability of renamable Horn formulas is feasible in polynomial time [Lew78]. A renamable Horn formula is unsatisfiable if and only if we can derive the empty clause from it by unit propagation; also, whenever we can derive from a formula the empty clause by means of unit resolution, then some unsatisfiable subset of the formula is renamable Horn [KBL99]. Thus RHORN lies in a certain sense half way between UP and HORN. Since RHORN is clause induced, both strong and deletion backdoor sets are of relevance. In contrast to HORN, not every strong RHORN-backdoor set is a deletion RHORN-backdoor set. Indeed, $\mathbf{b}_{\text{RHORN}}$ is a more general satisfiability parameter than $\mathbf{db}_{\text{RHORN}}$ as can be seen from Lemma 17.4.2 and the following example.

Example 17.4.3. For $1 \leq i \leq n$, let $F_i = \{\{x_i, y_i, z\}, \{x_i, \bar{y}_i, \bar{z}\}, \{\bar{x}_i, y_i\}, \{\bar{x}_i, \bar{y}_i\}\}$, and consider $F = \bigcup_{i=1}^n F_i$. Evidently $\{z\}$ is a strong RHORN-backdoor set of F , since each proper subset of $\{\{x_i, y_i\}, \{x_i, \bar{y}_i\}, \{\bar{x}_i, y_i\}, \{\bar{x}_i, \bar{y}_i\}\}$, $1 \leq i \leq n$, is renamable Horn. However, every deletion RHORN-backdoor set of F must contain at least one variable x_i or y_i for all $1 \leq i \leq n$. Hence $\mathbf{b}_{\text{RHORN}}(F) \leq 1$ and $\mathbf{db}_{\text{RHORN}}(F) \geq n$, which shows that $\mathbf{b}_{\text{RHORN}}$ is more general than $\mathbf{db}_{\text{RHORN}}$. \square

The detection of strong RHORN-backdoor sets is $W[1]$ -hard [GS12b], but the detection of deletion RHORN-backdoor sets is fixed-parameter tractable:

Theorem 17.4.7 ([RO08, GS12b]). *The problems $\mathbf{VER}(\mathbf{db}_{\text{RHORN}})$ and $\mathbf{SAT}(\mathbf{db}_{\text{RHORN}})$ are fixed-parameter tractable.*

Even the corresponding permissive problem is hard as well.

Theorem 17.4.8 ([GS12b]). *$\mathbf{SAT}(\mathbf{wb}_{\text{RHORN}})$ is $W[1]$ -hard.*

Boros et al. [BCH90] introduced an interesting class of CNF formulas, later called QHORN [BHS94], with favorable algorithmic properties: both recognition as well as deciding satisfiability of QHORN formulas can be performed in linear time. The class QHORN properly contains the fundamental classes RHORN and 2CNF:

$$\text{HORN} \subsetneq \text{RHORN} \subsetneq \text{QHORN} \supseteq 2\text{CNF}.$$

A CNF formula F is in QHORN if there is a certifying function $\beta : \text{var}(F) \cup \overline{\text{var}(F)} \rightarrow \{0, \frac{1}{2}, 1\}$ with $\beta(x) = 1 - \beta(\bar{x})$ for every $x \in \text{var}(F)$ such that $\sum_{l \in C} \beta(l) \leq 1$ for every clause C of F .

As QHORN generalizes RHORN, it is not surprising, that the problems $\mathbf{VER}(\mathbf{b}_{\text{QHORN}})$ and $\mathbf{VER}(\mathbf{wb}_{\text{QHORN}})$ are $W[2]$ -hard as well. However, Gaspers et al. [GOR⁺16] showed that the detection of deletion QHORN-backdoor sets is fixed-parameter approximable, which implies the fixed-parameter tractability of $\mathbf{SAT}(\mathbf{db}_{\text{QHORN}})$.

Theorem 17.4.9 ([GOR⁺16]). *The problem $\mathbf{SAT}(\mathbf{db}_{\text{QHORN}})$ is fixed-parameter tractable.*

This result was later improved by Ramanujan and Saurabh [RS17] who developed a general algorithmic framework for certain skew symmetric cut problems in graphs, and applied their method to show fixed-parameter tractability of $\mathbf{VER}(\mathbf{db}_{\text{QHORN}})$.

Table 17.1. The parameterized complexity of the detection of weak, strong, and deletion \mathcal{C} -backdoor sets, i.e., $\mathbf{VER}(\mathbf{wb}_{\mathcal{C}})$, $\mathbf{VER}(\mathbf{b}_{\mathcal{C}})$, and $\mathbf{VER}(\mathbf{db}_{\mathcal{C}})$, respectively, for various base classes \mathcal{C} .

Base Class	$\mathbf{VER}(\mathbf{wb}_{\mathcal{C}})$ weak	$\mathbf{VER}(\mathbf{b}_{\mathcal{C}})$ strong	$\mathbf{VER}(\mathbf{db}_{\mathcal{C}})$ deletion
$\mathcal{C} \in \text{Schaefer}$	W[2]-h ^[NRS04] (FPT)	FPT ^[NRS04]	FPT ^[NRS04]
$\mathcal{C} \in \text{Subsolver}$	W[P]-c ^[Sze05]	W[P]-c ^[Sze05]	n/a
FOREST	W[2]-h ^[GS12a] (FPT ^[GS12a])	? [†] (?)	FPT
RHORN	W[2]-h	W[2]-h (?)	FPT ^[RO08]
QHORN	W[2]-h ^[GOR⁺16]	W[2]-h ^[GOR⁺16] (?)	FPT ^[RS17]
CLU	W[2]-h ^[NRS07] (FPT)	W[2]-h ^[NRS07] (FPT ^[NRS07])	FPT ^[NRS07]

([∘]) It is indicated in parentheses if the complexity of the problem for 3CNF formulas is different from general CNF or unknown.

? It is open whether the problem is fixed-parameter tractable.

[†] Theorem 17.4.11 shows that the problem is fixed-parameter approximable.

^{n/a} Deletion backdoor sets are undefined for base classes that are not clause-induced.

Table 17.1, which is adapted from a survey paper [GS12b], gives an overview of parameterized complexity results of backdoor set detection problems for various base classes.

17.4.2. Heterogeneous Base Classes

One can enhance the power of a strong backdoor set B of a formula F , by allowing that for different assignments τ to B , $F[\tau]$ belongs to different base classes. This enhancement can be expressed in terms of *heterogeneous base classes* which are the union of individual base classes.

Example 17.4.4. Consider the following CNF formula $F_n = \{C, D_1, \dots, D_n\}$ where $C = \{x, \neg a_1, \dots, \neg a_n\}$ and $D_i = \{\neg x, b_i, c_i\}$. It is easy to see that any strong HORN-backdoor set needs to contain at least one of the variables b_i or c_i from each clause D_i , hence such a backdoor set must be of size $\geq n$; on the other hand, any strong 2CNF-backdoor set must contain at least $n - 2$ variables from the clause C ; However, $F_n[x = 0] \in \text{HORN}$ and $F_n[x = 1] \in \text{2CNF}$, hence the singleton $\{x\}$ is a strong $\text{HORN} \cup \text{2CNF}$ -backdoor set, where $\text{HORN} \cup \text{2CNF}$ is the heterogeneous base class consisting of all Horn and all 2CNF formulas. Note that any $F \in \text{HORN} \cup \text{2CNF}$ contains either only Horn clauses or only 2CNF clauses, not a mixture of both. \square

Weak backdoor sets with respect to heterogeneous base classes can also be considered. Identifying a base class with a class of instances that are solvable by a particular polynomial-time subsolver, one can consider a heterogeneous base class as a “portfolio subsolver,” where for each instance the best suitable subsolver from the portfolio is chosen.

The concept of heterogeneous base classes was introduced by Gaspers et al. [GMO⁺17] who studied the parameterized complexity of backdoor detection with respect to heterogeneous base classes in the context of SAT and CSP. For SAT,

they gave a full classification of the problem for heterogeneous base classes that are composed of base classes from Schaefer. For a nonempty subset $\mathcal{S} \subseteq \text{Schaefer}$, let $\mathcal{S}^\cup = \bigcup_{\mathcal{C} \in \mathcal{S}} \mathcal{C}$. It turned out that the detection of strong \mathcal{S}^\cup -backdoor sets is fixed-parameter tractable for all nonempty subsets $\mathcal{S} \subseteq \text{Schaefer}$ that do not contain any of the four pairs $\{\text{HORN}, \text{HORN}^-\}$, $\{0\text{-VAL}, 1\text{-VAL}\}$, $\{\text{HORN}, 1\text{-VAL}\}$, $\{0\text{-VAL}, \text{HORN}^-\}$, and is $W[2]$ -hard otherwise. Thus, from the 31 nonempty subsets of Schaefer, the problem is fixed-parameter tractable for 13 of them, and $W[2]$ -hard for the remaining 18. The fixed-parameter tractable cases can also be described in terms of the two maximal subsets $\{2\text{CNF}, \text{HORN}, 0\text{-VAL}\}$ and $\{2\text{CNF}, \text{HORN}^-, 1\text{-VAL}\}$:

Theorem 17.4.10 ([GMO⁺17]). *Let $\emptyset \neq \mathcal{S} \subseteq \text{Schaefer}$. If $\mathcal{S} \subseteq \{2\text{CNF}, \text{HORN}, 0\text{-VAL}\}$ or $\mathcal{S} \subseteq \{2\text{CNF}, \text{HORN}^-, 1\text{-VAL}\}$, then $\mathbf{VER}(\mathbf{b}_{\mathcal{S}^\cup})$, and consequently $\mathbf{SAT}(\mathbf{b}_{\mathcal{S}^\cup})$, are fixed-parameter tractable. Otherwise, $\mathbf{VER}(\mathbf{b}_{\mathcal{S}^\cup})$ is $W[2]$ -hard.*

Concerning the detection of weak backdoor sets, the $W[2]$ -hardness for individual base classes from Schaefer propagates to heterogeneous base classes that are composed from subsets of Schaefer [GMO⁺17].

17.4.3. Acyclic Formulas

Many NP-hard problems can be solved in polynomial time for problem instances that are in a certain sense acyclic. The satisfiability problem is no exception. There are various ways of defining a CNF formula to be acyclic. Here we consider acyclicity based on (undirected) incidence graphs (see Section 17.2.3). Let FOREST denote the class of CNF formulas whose undirected incidence graphs are forests. It is well known that the satisfiability problem can be solved in polynomial time for acyclic formulas [FMR08, SS10a]. The detection of deletion FOREST-backdoor sets is fixed-parameter tractable as one can use variants of algorithms for finding feedback vertex sets (also known as cycle cut sets) in graphs. It is not known, whether the detection of strong FOREST-backdoors is fixed-parameter tractable. However, Gaspers and Szeider [GS12a] showed that the problem is fixed-parameter approximable. They gave a fixed-parameter algorithm that either outputs a strong FOREST-backdoor set of size $\leq 2^k$, or correctly decides that the given formula has no strong FOREST-backdoor set of size $\leq k$. This result renders $\mathbf{SAT}(\mathbf{b}_{\text{FOREST}})$ fixed-parameter tractable. The detection of weak FOREST-backdoor sets is $W[2]$ -hard, but fixed-parameter tractable if the input formula is in 3CNF [GS12a].

Theorem 17.4.11 ([GS12a]). *The problem $\mathbf{SAT}(\mathbf{b}_{\text{FOREST}})$ is fixed parameter tractable.*

The class BAC of β -acyclic formulas, which was studied by Paulusma et al. [OPS13], is another possible base class defined in terms of acyclicity. A CNF formula is β -acyclic if its incidence graph $G^*(F)$ is chordal bipartite, which means that $G^*(F)$ has no induced cycle on six or more vertices (a cycle is induced if there are no edges between non-consecutive vertices). SAT-decision, as well as recognition, are polynomial-time solvable problems for BAC, which is a clause-induced

base class. The detection of strong BAC-backdoor sets is $W[2]$ -hard, but it is open whether the detection of deletion BAC-backdoor sets is fixed-parameter tractable [OPS13]. The $W[2]$ -hardness of the detection of weak BAC-backdoor sets can be established by instantiating the above mentioned generic reduction [GS12b, Proposition 1].

17.4.4. Hitting Formulas

Iwama [Iwa89] observed that one could determine the number of models of a CNF formula in polynomial time if any two clauses of the formula clash; such formulas are known as *hitting formulas* [KZ01]. Consider a hitting formula F with n variables. If a total truth assignment $\tau : \text{var}(F) \rightarrow \{0, 1\}$ does *not* satisfy a clause $C \in F$, it satisfies all other clauses of F . Hence we can count the total truth assignments that do not satisfy F by considering one clause after the other, and the number of models is therefore exactly $2^n - \sum_{C \in F} 2^{n-|C|}$. Of course, if a formula is a variable-disjoint union of hitting formulas—we call such a formula a *cluster formula*—we can still compute the number of models in polynomial time by taking the product of the number of models for each component. Since satisfiability (and obviously recognition) of cluster formulas can be established in polynomial time, the class CLU of cluster formulas is a base class. CLU is evidently clause induced.

Nishimura, Ragde, and Szeider considered the parameterized problem of detecting CLU-backdoor sets.

Theorem 17.4.12 ([NRS07]). **VER**(\mathbf{b}_{CLU}) is $W[2]$ -hard but **VER**(\mathbf{db}_{CLU}) is fixed-parameter tractable.

The hardness result is obtained by an fpt-reduction from the parameterized hitting set problem **HS**. The FPT result is achieved employing an algorithm that systematically destroys certain obstructions that consist of pairs or triples of clauses. To this end, the *obstruction graph* of a CNF formula F is considered. The vertex set of this graph is the set of variables of F ; two variables x, y are joined by an edge if and only if at least one of the following conditions hold:

1. F contains two clauses C_1, C_2 that do not clash, $x \in \text{var}(C_1 \cap C_2)$, and $y \in \text{var}(C_1 \setminus C_2)$;
2. F contains three clauses C_1, C_2, C_3 such that C_1 and C_3 do not clash, $x \in \text{var}((C_1 \setminus C_3) \cap \overline{C_2})$, and $y \in \text{var}((C_3 \setminus C_1) \cap \overline{C_2})$.

Since the vertex covers of the obstruction graph are exactly the deletion CLU-backdoor sets, we can find smallest deletion CLU-backdoor sets by using a vertex cover algorithm. Hence **VER**($\mathbf{db}_{\text{CLU}}(F)$) is fixed-parameter tractable.

Example 17.4.5. Consider formula $F = \{\{u, v\}, \{s, \bar{u}, \bar{v}\}, \{u, \bar{v}, w, \bar{r}\}, \{r, \bar{w}, x, y\}, \{\bar{x}, y, z\}, \{\bar{y}, \bar{z}\}, \{\bar{s}, \bar{t}\}, \{\bar{t}\}, \{t, w\}\}$. The obstruction graph has the edges $rw, st, tw, uw, vw; ru, rv, rx, ry, su, sv, wx, wy$. The set $B = \{r, s, w\}$ forms a vertex cover of the obstruction graph; there is no vertex cover of size two. B is a deletion CLU-backdoor set and consequently also a strong CLU-backdoor set of F . There is, however, the smaller strong CLU-backdoor set $B' = \{w, s\}$. \square

The algorithm of Nishimura et al. outlined above can be used to count the number $\#(F)$ of models of a given formula F . More generally, assume that we have a base class \mathcal{C} such that $\#(F)$ can be computed in polynomial time for every $F \in \mathcal{C}$ (which is the case for CLU). Then, if we have a strong \mathcal{C} -backdoor set B of an arbitrary CNF formula F , we can compute $\#(F)$ utilizing the identity

$$\#(F) = \sum_{\tau: B \rightarrow \{0,1\}} 2^{d(F,\tau)} \#(F[\tau])$$

where $d(F, \tau) = |\text{var}(F - B) \setminus \text{var}(F[\tau])|$ denotes the number of variables that disappear from $F[\tau]$ without being instantiated. Thus determining $\#(F)$ reduces to determining the number of models for $2^{|B|}$ formulas of the base class \mathcal{C} . In particular, the above considerations yield a fixed-parameter algorithm for model counting parameterized by the clustering-width. Note, however, that for the classes HORN and 2CNF, the model counting problem is $\#P$ -hard (even for monotone formulas) [Rot96]. Thus knowing a small strong backdoor set with respect to these classes does not help to count the number of models efficiently.

We will see two further satisfiability parameters that generalize hitting formulas in Section 17.5.2 (conflict treewidth) and Section 17.6.2 (h-modularity).

17.4.5. Empty Clause Detection

Dilkina, Gomes, and Sabharwal [DGS07] suggested strengthening the concept of strong backdoor sets by means of *empty clause detection*. Let \mathcal{E} denote the class of all CNF formulas that contain the empty clause. For a base class \mathcal{C} we put $\mathcal{C}^{\{\}} = \mathcal{C} \cup \mathcal{E}$; we call $\mathcal{C}^{\{\}}$ the base class obtained from \mathcal{C} by adding *empty clause detection*. Formulas can have much smaller strong $\mathcal{C}^{\{\}}$ -backdoor sets than strong \mathcal{C} -backdoor sets; Dilkina et al. give empirical evidence for this phenomenon considering various base classes. Note that the addition of empty clause detection makes only sense for strong backdoor sets [DGS07], not for weak or deletion backdoor sets. Dilkina et al. show that given a CNF formula F and an integer k , determining whether F has a strong $\text{HORN}^{\{\}}$ -backdoor set of size k is both NP-hard and co-NP-hard (here k is considered only as part of the input and not as a parameter). Thus, the non-parameterized complexity of the search problem for strong HORN-backdoor sets gets harder when empty clause detection is added. Also, the parameterized complexity gets harder, which can be shown using results from Fellows et al. [FSW06].

Theorem 17.4.13 ([Sze08]). *For $\mathcal{C} \in \{\text{HORN}^{\{\}}, 2\text{CNF}^{\{\}}, \text{RHORN}^{\{\}}\}$ the problem $\text{VER}(\mathbf{b}_{\mathcal{C}})$ is $W[1]$ -hard.*

17.5. Treewidth

Treewidth is an important graph invariant that measures the “tree-likeness” of a graph. Many otherwise NP-hard graph problems such as Hamiltonicity and 3-colorability are fixed-parameter tractable if parameterized by the treewidth of the input graph. It is generally believed that many practically relevant problems do have low treewidth [Bod93]. For taking the treewidth of a CNF formula, one

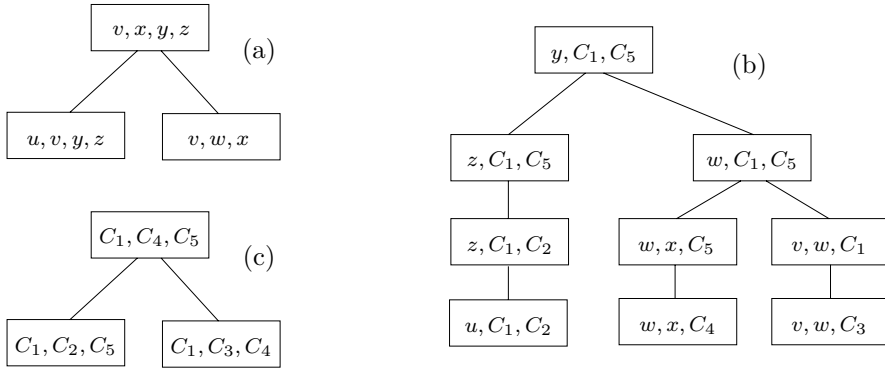


Figure 17.3. Tree decompositions of the primal graph (a), the incidence graph (b), and the dual graph (c)

needs to represent the structure of a formula as a graph. For this, we can use the graphs discussed in Section 17.2.3

Tree decompositions of graphs and the associated parameter treewidth were studied by Robertson and Seymour in their famous Graph Minors Project. A *tree decomposition* of a graph $G = (V, E)$ is a tree $T = (V', E')$ together with a labeling function $\chi : V' \rightarrow 2^V$ associating to each tree node $t \in V'$ a bag $\chi(t)$ of vertices in V such that the following three conditions hold:

1. every vertex in V occurs in some bag $\chi(t)$;
2. for every edge $xy \in E$ there is a bag $\chi(t)$ that contains both x and y ;
3. if $\chi(t_1)$ and $\chi(t_2)$ both contain x , then each bag $\chi(t_3)$ contains x if t_3 lies on the unique path from t_1 to t_2 .

The *width* of a tree decomposition is $\max_{t \in V'} |\chi(t)| - 1$. The *treewidth* of a graph is the minimum width over all its tree decompositions. The treewidth of a graph is a measure for its acyclicity, i.e., the smaller the treewidth, the less cyclic the graph. In particular, a graph is *acyclic* if and only if it has treewidth 1.

The above definition of a tree decomposition can be easily generalized to hypergraphs by requiring in item (2) that all vertices in each hyperedge occur together in some bag. Every tree decomposition of the primal graph $G(F)$ of a CNF formula F is a tree decomposition of the hypergraph $H(F)$. Thus, if the treewidth of the primal graph is k , the cardinality of each clause of F cannot be larger than $k + 1$.

For a CNF formula F , we introduce the following notions of treewidth: the (*primal*) *treewidth* \mathbf{tw} of F is the treewidth of its primal graph $G(F)$, the *incidence treewidth* \mathbf{tw}^* of F is the treewidth of its incidence graph $G^*(F)$, and the *dual treewidth* \mathbf{tw}^d of F is the treewidth of its dual graph $G^d(F)$. Tree decompositions of the three graphs associated with formula F in Figure 17.1 are shown in Figure 17.3. Since there are no tree decompositions of these graphs of smaller width, we know that $\mathbf{tw}(F) = 3$ and $\mathbf{tw}^*(F) = \mathbf{tw}^d(F) = 2$.

Kolaitis and Vardi [KV00] have shown that always $\mathbf{tw}^*(F) \leq \mathbf{tw}(F) + 1$ and $\mathbf{tw}^*(F) \leq \mathbf{tw}^d(F) + 1$. In other words, the incidence treewidth dominates

the primal treewidth and the dual treewidth. On the other hand, there exist families of CNF formulas with incidence treewidth one and arbitrarily large primal treewidth and dual treewidth, i.e., this domination is strict.

Example 17.5.1. Consider the two families $F_n = \{\{x_1, x_2, \dots, x_n\}\}$ and $G_n = \{\{x_1, y\}, \{x_2, y\}, \dots, \{x_n, y\}\}$ of CNF formulas. Then $\mathbf{tw}^*(F_n) = \mathbf{tw}^*(G_n) = 1$ while $\mathbf{tw}(F_n) = \mathbf{tw}^d(G_n) = n - 1$. \square

The intuitive idea of tree decompositions is to partition a graph into clusters of vertices that can be organized as a tree. The smaller the width of a tree decomposition, the more efficiently we can decide satisfiability of the corresponding CNF formula by a bottom-up dynamic programming approach on the tree decomposition. Thus, we aim to construct a tree decomposition of width as small as possible; in the optimal case, the width of the tree decomposition equals the treewidth of the graph.

In general, computing the treewidth of a graph is NP-hard [ACP87]. However, since tree decompositions with large width do not help us in deciding satisfiability efficiently, we are more interested in graphs with small treewidth. Bodlaender [Bod96] has shown that it can be decided in linear time whether the treewidth of a graph is at most k if k is a constant. This immediately implies fixed-parameter tractability of the problems $\mathbf{VER}(\mathbf{tw})$, $\mathbf{VER}(\mathbf{tw}^*)$, and $\mathbf{VER}(\mathbf{tw}^d)$. In Section 17.5.4 we will review algorithms for constructing tree decompositions.

17.5.1. Deciding Satisfiability

As mentioned above, if a tree decomposition of the primal graph, the incidence graph, or the dual graph is given, we can decide satisfiability of the corresponding CNF formula by a bottom-up dynamic programming approach on the tree decomposition. The smaller the width of the given tree decomposition, the more efficiently we can decide satisfiability. In particular, from Yannakakis's algorithm [Yan81] we obtain the following result as already observed by Gottlob et al. [GSS02].

Theorem 17.5.1. *The problem $\mathbf{SAT}(\mathbf{tw})$ is fixed-parameter tractable.*

To see this, consider a tree decomposition of the primal graph of a given CNF formula F and let k be the width of this tree decomposition. Note that the number of nodes of the tree can be bounded by the length $n = \|F\|$. Now, we associate with each node t of the tree a table M_t with $|\chi(t)|$ columns and at most $2^{|\chi(t)|}$ rows. Each row contains Boolean values encoding a truth assignment to the variables in $\chi(t)$ that does not falsify any clause of F . The size of each table is therefore bounded by $2^{k+1}(k+1)$ and all such tables can be computed in time $\mathcal{O}(2^k kn^2)$. In this way we can transform our SAT problem into an equivalent constraint satisfaction problem by a fixed-parameter algorithm with parameter treewidth. This constraint satisfaction problem can now be solved by Yannakakis's algorithm in time $\mathcal{O}(4^k kn)$. Yannakakis's algorithm works as follows: for each node t of the tree it is checked whether to each truth assignment in table $M_{t'}$ associated with t 's parent t' there exists a consistent truth assignment in table M_t . We remove truth assignments in table $M_{t'}$ to which no such consistent truth assignment

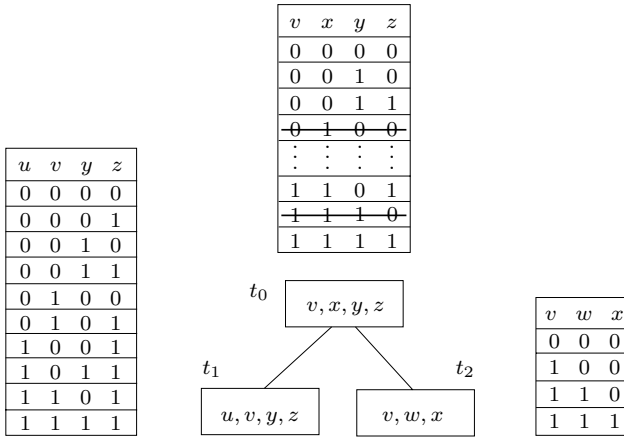


Figure 17.4. A fixed-parameter algorithm for $\text{SAT}(\text{tw})$

in table M_t exists. The whole procedure works in a bottom-up manner, i.e., a node is processed if all of its children have already been processed. The CNF formula F is satisfiable if and only if some truth assignments are left in the table associated with the root after termination of this procedure. Thus, in summary, we can decide $\text{SAT}(\text{tw})$ in time $\mathcal{O}^*(4^k)$. By using an improved algorithm, we can decide $\text{SAT}(\text{tw})$ in time $\mathcal{O}^*(2^k)$ [SS10a].

Example 17.5.2. Consider the primal graph of the CNF formula F in Figure 17.1 and its tree decomposition in Figure 17.3(a). The tables associated with each tree node are shown in Figure 17.4: there are 14 truth assignments in table M_{t_0} associated with the root t_0 , 10 in table M_{t_1} associated with the left child t_1 , and 4 in table M_{t_2} associated with the right child t_2 . Now let us start with the left child t_1 and remove the rows 1010 and 1110 from table M_{t_0} since there are no consistent truth assignments in table M_{t_1} . Then we consider the right child t_2 and remove the rows 0100, 0101, 0110, and 0111 from table M_{t_0} since there are no consistent truth assignments in table M_{t_2} . Since there are no further nodes to be processed, we are finished and know that F is satisfiable. \square

Since tw^* strictly generalizes tw , the following result is stronger than Theorem 17.5.1.

Theorem 17.5.2. *The problem $\text{SAT}(\text{tw}^*)$ is fixed-parameter tractable.*

Since incidence treewidth strictly dominates primal treewidth and dual treewidth as already mentioned above, this result implies both Theorem 17.5.1 and fixed-parameter tractability of $\text{SAT}(\text{tw}^d)$. The situation is different for “generalized satisfiability” also known as “Boolean constraint satisfaction” where Boolean relations replace clauses. Generalized satisfiability is fixed-parameter tractable for the parameter primal treewidth but $W[1]$ -hard for the parameter incidence treewidth [SS10b].

In the following, we will discuss three approaches to establishing Theorem 17.5.2.

Proof via a Logic Meta Theorem Courcelle has shown that every graph property that can be expressed in a certain formalism (monadic second-order logic, MSO) can be decided in linear time for graphs of bounded treewidth [Cou88]. This theorem applies to many NP-hard graph properties such as 3-colorability and yields fixed-parameter tractability for these problems with respect to parameter treewidth. Thus MSO theory provides a very general and convenient tool for classifying problems parameterized by treewidth as fixed-parameter tractable. Using the general methods of MSO theory, one can easily establish fixed-parameter tractability of $\mathbf{SAT}(\mathbf{tw}^*)$ [CMR01, GS08, Sze04b]. However, the algorithms obtained via the generic constructions are impractical.

Proof via Clause Splitting It is well known (see, e.g., [GJ79, p. 48]) that a CNF formula F can be transformed in polynomial time into an equisatisfiable 3CNF formula F_3 , by repeatedly splitting a clause $(\ell_1 \vee \ell_2 \vee \ell_3 \vee \dots \vee \ell_r)$ with $r \geq 4$ into two clauses $(\ell_1 \vee \ell_2 \vee x)$ and $(\neg x \vee \ell_3 \vee \dots \vee \ell_r)$ where x is a new variable. Samer and Szeider [SS10b, Remark, pp. 111] have shown that in general this procedure can increase the incidence treewidth arbitrarily, but if applied properly (respecting an ordering of clauses and variables that is inferred from a tree decomposition of the incidence graph), the incidence treewidth stays bounded.

Lemma 17.5.3 (Splitting Lemma [SS10b]). *Given a CNF formula F together with a tree decomposition of width k of the incidence graph of F . By splitting clauses we can obtain in polynomial time an equisatisfiable 3CNF formula with incidence treewidth at most $k + 1$ and primal treewidth at most $3(k + 1)$.*

Thus, there is an fpt-reduction from $\mathbf{SAT}(\mathbf{tw}^*)$ to $\mathbf{SAT}(\mathbf{tw})$, and hence Theorem 17.5.1 implies Theorem 17.5.2.

Proof via Dynamic Programming For more practical algorithms, however, one needs to use more closely the combinatorial structure of the particular problem at hand. Fischer et al. [FMR08] and Samer and Szeider [SS10a] presented practical fixed-parameter algorithms for the more general problem $\#\mathbf{SAT}(\mathbf{tw}^*)$ of counting the number of models. This trivially implies Theorem 17.5.2, since a CNF formula is satisfiable if and only if it has at least one model. In the following we present the algorithm introduced by Samer and Szeider. The algorithm is based on “nice” tree decompositions, which are a special kind of tree decompositions. It is well known that one can transform any tree decomposition of width k in linear time into a nice tree decomposition of width at most k [BK96, Klo94].

For each node t , we write X_t and F_t to denote the set of all variables and clauses occurring in $\chi(t')$, respectively, for some node t' in the subtree rooted at t . Moreover, we use the shorthands $\chi_v(t) = \chi(t) \cap X_t$ and $\chi_c(t) = \chi(t) \cap F_t$ for the set of variables and the set of clauses in $\chi(t)$ respectively. For each truth assignment $\alpha : \chi_v(t) \rightarrow \{0, 1\}$ and subset $A \subseteq \chi_c(t)$, we define $N(t, \alpha, A)$ as the set of truth assignments $\tau : X_t \rightarrow \{0, 1\}$ for which the following two conditions hold:

1. $\tau(x) = \alpha(x)$ for all variables $x \in \chi_v(t)$ and
2. A is exactly the set of clauses of F_t that are not satisfied by τ .

Now, we associate with each node t of the tree a table M_t with $|\chi(t)| + 1$ columns and $2^{|\chi(t)|}$ rows. The first $|\chi(t)|$ columns contain Boolean values encod-

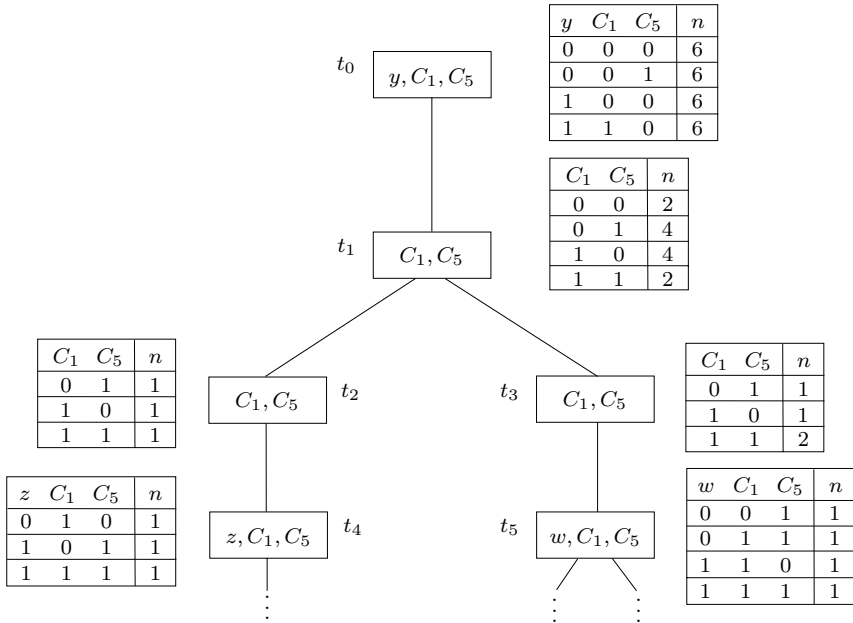


Figure 17.5. A fixed-parameter algorithm for $\#\text{SAT}(\text{tw}^*)$

ing $\alpha(x)$ for variables $x \in \chi_v(t)$, and membership of C in A for clauses $C \in \chi_c(t)$. The last column contains the integer $n(t, \alpha, A) = |N(t, \alpha, A)|$. Given the tables of the children of some node t , the table M_t can be computed in time $\mathcal{O}(4^k kl)$, where l is the cardinality of the largest clause. All the tables associated with tree nodes can be computed in a bottom-up manner. The number of models of the corresponding CNF formula F is then given by $\sum_{\alpha: \chi_v(r) \rightarrow \{0,1\}} n(r, \alpha, \emptyset)$, where r is the root of the tree. Thus, we can decide $\#\text{SAT}(\text{tw}^*)$ in time $\mathcal{O}^*(4^k)$. Based on an observation by Björklund [Bjö18], Slivovsky and Szeider [SS20] improved the running time for $\#\text{SAT}(\text{tw}^*)$ to $\mathcal{O}^*(2^k)$ through efficiently computing *covering products* [BHKK07].

Example 17.5.3. Consider the incidence graph of the CNF formula F in Figure 17.1 and its tree decomposition in Figure 17.3(b). A fragment of the corresponding nice tree decomposition and the tables associated with each tree node are shown in Figure 17.5. Note that we omit for simplicity those rows from the tables where $n = 0$. We assume that the tables M_{t_4} and M_{t_5} associated with the nodes t_4 and t_5 respectively have already been computed in a bottom-up manner starting from the leaves. For example, the entries in table M_{t_4} mean that (i) there exists exactly one truth assignment $\tau : X_{t_4} \rightarrow \{0, 1\}$ such that $\tau(z) = 0$ and τ satisfies all clauses in F_{t_4} except C_1 , (ii) there exists exactly one truth assignment $\tau : X_{t_4} \rightarrow \{0, 1\}$ such that $\tau(z) = 1$ and τ satisfies all clauses in F_{t_4} except C_5 , and (iii) there exists exactly one truth assignment $\tau : X_{t_4} \rightarrow \{0, 1\}$ such that $\tau(z) = 1$ and τ satisfies all clauses in F_{t_4} except C_1 and C_5 . The next step is to compute the tables M_{t_2} and M_{t_3} from tables M_{t_4} and M_{t_5} respectively.

Since t_2 and t_3 are forget nodes (the variable z has been forgotten in t_2 and the variable w has been forgotten in t_3), this can be done according to the rule for forget nodes as given in [SS10a]. Now we compute table M_{t_1} from tables M_{t_2} and M_{t_3} according to the rule for join nodes. Finally, we compute table M_{t_0} from table M_{t_1} according to the rule for introduce nodes. From table M_{t_0} we can now see that there are exactly 12 truth assignments $\tau : X_{t_0} \rightarrow \{0, 1\}$ such that τ satisfies all clauses in F_{t_0} (for 6 of these truth assignments it holds that $\tau(y) = 0$ and for 6 of them it holds that $\tau(y) = 1$), where $X_{t_0} = \text{var}(F)$ and $F_{t_0} = F$. Consequently, the CNF formula F has exactly 12 models. \square

Bacchus, Dalmao, and Pitassi [BDP03] presented another fixed-parameter algorithm for computing the number of models of a CNF formula F . The parameter in their algorithm is the *branchwidth* of the hypergraph $H(F)$. Similar to tree decompositions, branch decompositions and the corresponding branchwidth were introduced by Robertson and Seymour in their Graph Minors Project. It is well known that a graph with treewidth k has branchwidth at most $k + 1$ and that a graph with branchwidth k has treewidth at most $3k/2$ [RS91]. Thus, primal treewidth and branchwidth are domination equivalent satisfiability parameters. Bacchus et al. define a static ordering of the variables of F based on the branch decomposition of $H(F)$ and run a DPLL procedure with caching on this ordering. In particular, they decompose the input formula and intermediate formulas into disjoint components; these components are cached when they are solved the first time, which allows truncating the search-tree of the DPLL procedure. The resulting algorithm runs in time $2^{\mathcal{O}(k)}n^c$, where k is the branchwidth, n is the number of variables, and c is a constant.

17.5.2. Consensus Treewidth and Conflict Treewidth

Recall from above the definitions of the consensus graph $G^{\text{cs}}(F)$ and the conflict graph $G^{\text{c}}(F)$, associated with a CNF formula F . We define the *consensus treewidth* $\text{tw}^{\text{cs}}(F) = \text{tw}(G^{\text{cs}}(F))$ and the *conflict treewidth* $\text{tw}^{\text{c}}(F) = \text{tw}(G^{\text{c}}(F))$. Ganian and Szeider [GS17] introduced these two satisfiability parameters and studied their parameterized complexity. The consensus treewidth turned out to be an interesting parameter which is not dominated by any known satisfiability parameter and admits fixed-parameter tractability.

Theorem 17.5.4 ([GS17]). *The problem $\#\text{SAT}(\text{tw}^{\text{cs}})$ is fixed-parameter tractable.*

On the other hand, conflict treewidth seems to be not that useful:

Theorem 17.5.5 ([GS17]). *The problem $\text{SAT}(\text{tw}^{\text{c}})$ is W[1]-hard.*

W[1]-hardness is avoided by bounding the size of clauses and eliminating pure literals. For such formulas, however, conflict treewidth is dominated by incidence treewidth.

17.5.3. Expression Treewidth

Before discussing algorithms for computing tree decompositions, let us briefly mention a very general approach to applying treewidth to CNF formulas (or rather

Boolean functions in general), that was suggested by Jha and Suciú [JS12]. They define the *expression treewidth* of a Boolean function f as the smallest treewidth of any Boolean circuit representing the function. The circuit is considered as a directed acyclic graph D , and its treewidth is taken from the undirected graph \underline{D} obtained from D by ignoring the orientation of edges. By letting f_F denote the Boolean function represented by a CNF formula F , we can define the satisfiability parameter expression treewidth $\mathbf{xtw}(F)$ as the expression treewidth of f_F . Expression treewidth strictly dominates incidence treewidth [JS12]. Courcelle's meta-theorem [Cou88] can be used to show that $\mathbf{SAT}(\mathbf{xtw})$ is fixed-parameter tractable if the circuit D with minimal treewidth is provided as the input. Bova and Szeider [BS17] have shown that $\mathbf{VER}(\mathbf{xtw})$ is at least decidable, using a meta-theorem by Seese [See91]. It would be interesting to know more efficient algorithms for this problem.

17.5.4. Tree Decomposition Algorithms

As for most algorithms, also in the case of computing tree decompositions, there has to be a tradeoff made between runtime, space requirement, and simplicity. In the following, we use n to denote the number of vertices of the given graph. The current fastest exact tree decomposition algorithm runs in time $\mathcal{O}^*(1.8899^n)$ and is due to Fomin, Kratsch, and Todinca [FKT04] and Villanger [Vil06]. This algorithm is based on the computation of potential maximal cliques. Bodlaender et al. [BFK⁺06] developed a simpler algorithm based on a recursive divide-and-conquer technique that requires polynomial space and runs in time $\mathcal{O}^*(4^n)$. For special classes of graphs, however, there exist exact tree decomposition algorithms that run in polynomial (or even linear) time [Bod93].

Polynomial-time algorithms also exist in the case of bounded treewidth. These algorithms are fixed-parameter algorithms: Reed's algorithm [Bod93, Ree92] decides in time $\mathcal{O}(n \log n)$ whether the treewidth of a graph is at most k and, if so, computes a tree decomposition of width at most $3k + 2$. Bodlaender and Kloks [BK96] developed an algorithm with the same asymptotic runtime as Reed's algorithm that decides whether the treewidth of a graph is at most k and, if so, computes a tree decomposition of width at most k . Bodlaender [Bod96] improved this result to a linear-time algorithm. The hidden constant factors in the runtime of the latter two algorithms, however, are very large so that they are only practical for very small k (e.g., up to $k = 5$) [BK96, Bod05].

Algorithms that approximate treewidth by finding tree decompositions of *nearly* minimal width give a guarantee on the quality of the output. Bodlaender et al. [BDD⁺16] gave a single exponential algorithm that runs in $2^{\mathcal{O}(k)}n$ time, and either outputs a tree decomposition of the given graph G of width at most $5k + 4$, or determines that $\text{tw}(G) > k$.

In practice, it often suffices to obtain tree decompositions of small width without any guarantees. There exist several powerful tree decomposition heuristics for this purpose. In the worst case, the width of tree decompositions obtained by such heuristics can be far from treewidth; however, their width is often small in practically relevant cases. An important class of tree decomposition heuristics is based on finding an appropriate linear ordering of the vertices from which

a tree decomposition can be constructed [Bod05]. *Minimum degree* and *minimum fill-in* [Bod05], *lexicographic breadth-first search* [RTL76], and *maximum cardinality search* [TY84] are well-known examples of such ordering heuristics. Koster, Bodlaender, and van Hoesele [KBvH01a, KBvH01b] compared several tree decomposition heuristics by empirical evaluation.

We refer the interested reader to Bodlaender's excellent survey papers [Bod93, Bod05] for a more extensive overview of tree decomposition algorithms.

One can also use SAT solvers to determine the treewidth of graphs. Samer and Veith [SV09] gave a first SAT encoding, that, given a graph G and an integer k , produces a CNF formula $F(G, k)$ which is satisfiable if and only if $\text{tw}(G) \leq k$. This approach was further improved [BJ14, BBE17], as well as used to improve a heuristically computed tree decomposition locally [FLS17].

17.5.5. Beyond Treewidth

Beside treewidth, other decomposition-based measures for the tractability of certain computation problems with graph and hypergraph representations have been proposed in the literature. One of the most prominent examples is *clique-width* [CER93]. Intuitively, the clique-width $\text{cwd}(G)$ of a graph G is the smallest number of colors required to construct the graph by means of certain operations that do not distinguish between vertices of the same color. Clique-width is defined for undirected graphs and directed graphs. Let \underline{D} be the undirected graph obtained from a directed graph D by ignoring the orientation of edges. Then $\text{cwd}(\underline{D}) \leq \text{cwd}(D)$.

Bounding the clique-width of primal graphs does not help with deciding satisfiability: one can easily make the primal graph a clique as follows: Let F be a CNF formula and x a new variable not occurring in F . Now consider the CNF formula F^* obtained from F by adding the two clauses $C = \text{var}(F) \cup \{x\}$ and $C' = \{\bar{x}\}$. Clearly F and F^* are equisatisfiable and even share the same number of models. Now the primal graph of F^* is a clique and so it has clique-width 2. For the incidence graph, however, bounding the clique-width helps, as we shall see below.

We distinguish between $\mathbf{cwd}^*(F) = \text{cwd}(G^*(F))$ and $\mathbf{dcwd}^*(F) = \text{cwd}(D^*(F))$. From results by Courcelle and Olariu [CO00] it follows that \mathbf{cwd}^* strictly dominates \mathbf{dcwd}^* , and \mathbf{dcwd}^* strictly dominates \mathbf{tw}^* . The problems $\mathbf{VER}(\mathbf{cwd}^*)$ and $\mathbf{VER}(\mathbf{dcwd}^*)$ are not known to be fixed-parameter tractable, but fixed-parameter approximable [Oum05, FMR08].

Theorem 17.5.6 ([OPS13]). *The problem $\mathbf{SAT}(\mathbf{cwd}^*)$ is $W[1]$ -hard.*

This hardness even holds if the corresponding decomposition is provided with the input.

Theorem 17.5.7 ([SS13]). *$\#\mathbf{SAT}(\mathbf{cwd}^*)$ is XP-tractable.*

This result is obtained by a dynamic programming algorithm in which certain projections of truth assignments play an important role. Such projections have already been used previously for showing XP-tractability of $\#\mathbf{SAT}$ for the satisfiability parameter *modular treewidth* [PSS16] which is the treewidth of the

incidence graph, taken after the contraction of modules (i.e., of vertices with the same neighbors). Modular treewidth strictly dominates treewidth and is, in turn, strictly dominated by clique-width. Sæther et al. [STV15] generalized the algorithm underlying Theorem 17.5.7 to be applicable to larger classes of formulas, however, requiring the corresponding decomposition to be provided with the input.

Theorem 17.5.8 ([CMR01, FMR08]). $\#\text{SAT}(\text{dcwd}^*)$ is fixed-parameter tractable.

The fixed-parameter tractability of $\#\text{SAT}(\text{dcwd}^*)$ follows from meta-theorem of monadic second-order logic [CMR01] similarly as in the case of treewidth. Also a direct dynamic programming algorithm is known [FMR08]. For computing clique-width decompositions of (directed) incidence graphs one can use SAT-encodings [HS15, Par16].

Rank-width [RS86] is a graph invariant that is similar to clique-width and can also be defined for directed and undirected incidence graphs. The corresponding satisfiability parameters are domination equivalent with directed and undirected clique-width, respectively. For satisfiability checking and model counting, directed rank-width has an advantage over dcwd^* since the corresponding verification problem is known to be fixed-parameter tractable and so one saves the approximation error. Moreover, even if optimal clique-width or rank-width decompositions are provided, the rank-width based algorithm can be exponentially faster than the clique-width based algorithm [GHO13].

Generalizations of treewidth like hypertree-width [GLS02], spread-cut width [CJG08], and fractional hypertree-width [GM06] are defined for hypergraphs in the context of constraint satisfaction and conjunctive database queries. According to the current status of knowledge, they have no relevance for the satisfiability problem of CNF formulas [SS10a]. This can be seen, using the simple construction we considered at the beginning of Section 17.5.5: the hypergraph $H(F^*)$ is acyclic [GLS02], and so hypertree-width, spread-cut width, and fractional hypertree-width of $H(F^*)$ are 1. A similar construction can be made with respect to the dual hypergraph $H^d(F)$ [SS10a].

17.6. Further Satisfiability Parameters

In this section, we will discuss further satisfiability parameters that are (i) based on a combination of backdoor sets and treewidth, (ii) based on a community structure in the formula, and (iii) based on matchings in the incidence graph of the formula.

17.6.1. Hybrid Satisfiability Parameters

The general approaches to satisfiability parameters as discussed in the previous two sections (based on backdoors and decompositions, respectively) are complementary. Take, for instances \mathbf{b}_{HORN} (size of a smallest strong HORN-backdoor) and tw^* (treewidth of the incidence graph). The two satisfiability parameters are incomparable, as one can construct Horn formulas of arbitrarily large incidence

treewidth (i.e., \mathbf{b}_{HORN} is 0 and \mathbf{tw} is unbounded), and formulas of arbitrarily large \mathbf{b}_{HORN} and bounded primal treewidth.

Several ways for combining the strengths of the two approaches have been considered which we will briefly discuss.

Strong Backdoors into Bounded Incidence Treewidth Gaspers and Szeider [GS13] considered strong $\mathcal{W}_{\leq t}$ -backdoor sets where $\mathcal{W}_{\leq t}$ consists of all CNF formulas F with $\mathbf{tw}^*(F) \leq t$, where $t > 0$ is a fixed constant. $\mathcal{W}_{\leq t}$ is suitable as a base class (Theorem 17.5.2), even $\#\text{SAT}$ is polynomial-time tractable for $\mathcal{W}_{\leq t}$. For $t = 1$ we obtain the base class **FOREST** discussed above. It is crucial to consider strong backdoor sets, not just deletion backdoor sets, as $\mathbf{db}_{\mathcal{W}_{\leq t}}$ is easily seen to be dominated by \mathbf{tw}^* , whereas $\mathbf{b}_{\mathcal{W}_{\leq t}}$ strictly dominates \mathbf{tw}^* [GS13]. When we know a strong $\mathcal{W}_{\leq t}$ -backdoor set of size k of a formula F , we can compute $\#(F)$ in $\mathcal{O}^*(2^k)$ time. The main problem is finding such a backdoor set, i.e., $\mathbf{VER}(\mathbf{b}_{\mathcal{W}_{\leq t}})$. Gaspers and Szeider [GS13] showed that this problem is fixed-parameter approximable, which implies the following result.

Theorem 17.6.1 ([GS13]). *For every $t \geq 1$, $\#\text{SAT}(\mathbf{b}_{\mathcal{W}_{\leq t}})$ is fixed-parameter tractable.*

By adjusting t and backdoor set size, one can fine-tune the algorithm behind Theorem 17.6.1 to a particular class of input formulas. Fomin et al. [FLM⁺15] considered the problems $\mathbf{SAT}(\mathbf{wb}_{\mathcal{W}_{\leq t}})$ and $\mathbf{SAT}(\mathbf{b}_{\mathcal{W}_{\leq t}})$ for the special case where the input is an r -CNF formula for an arbitrary constant r . They showed that the corresponding permissive problems are fixed-parameter tractable. Their algorithms avoid the computation of a fixed-parameter approximation and achieves a single exponential running time.

Backdoor Treewidth Next we consider an approach due to Ganian et al. [GRS17b, GRS17a], where one considers not the size of a smallest strong \mathcal{C} -backdoor set of a given CNF formula F as the parameter, but the treewidth of a strong \mathcal{C} -backdoor set of smallest treewidth. The treewidth of a strong \mathcal{C} -backdoor set B is taken in terms of a *torso graph* G_F^B . The vertices of G_F^B are the variables in B , and an edge connects two variables $x, y \in B$ if and only if the incidence graph $G^*(F)$ contains a path between x and y that does not traverse any vertex in X (except x and y). Thus, the torso graph is obtained by “collapsing” (possibly large parts of) the incidence graph to single edges. The *\mathcal{C} -backdoor treewidth* of F , denoted $\mathbf{tw}_{\mathcal{C}}(F)$, is the smallest $\mathbf{tw}(G_F^B)$ over all strong \mathcal{C} -backdoor sets B of F . Figure 17.6 shows an example for this construction. We note that, in general, a strong \mathcal{C} -backdoor set B of F with smallest $\mathbf{tw}(G_F^B)$ might not be among the smallest strong \mathcal{C} -backdoor sets of F . In the following we focus on base classes $\mathcal{C} \in \{\text{HORN}, \text{HORN}^{-1}, 2\text{CNF}\}$.

If we know a strong \mathcal{C} -backdoor set B with $\mathbf{tw}(G_F^B) = k$, we can compile connected components of $G^*(F) - B$ into constraints, and so represent F by an equisatisfiable Boolean CSP instance over the variables B , whose primal treewidth is k . Solving this CSP instance is fixed-parameter tractable parameterized by k [GSS02]. Hence again, the challenging task is to find the set B , which was shown by Ganian et al. [GRS17a] to be fixed-parameter tractable:

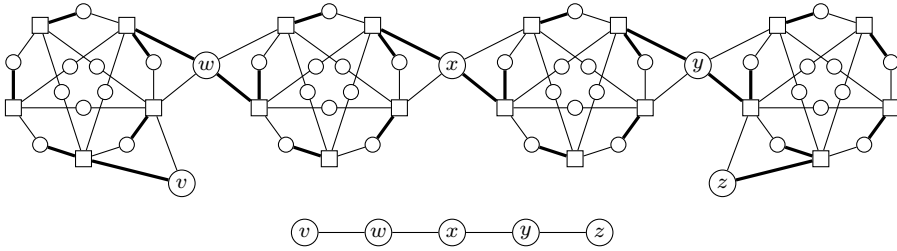


Figure 17.6. Top: an example of a formula F , drawn by its incidence graph, where positive occurrences of variables are indicated with bold edges. $B = \{v, w, x, y, z\}$ forms a strong HORN-backdoor set of F . Bottom: the torso graph G_F^B , which is a path and has, therefore, treewidth 1. Consequently, the HORN-backdoor treewidth is $\mathbf{b}_{\text{HORN}}(F) = 1$.

Theorem 17.6.2 ([GRS17a]). *For $\mathcal{C} \in \{\text{HORN}, \text{HORN}^-, 2\text{CNF}\}$, the problems $\text{VER}(\text{tw}_{\mathcal{C}})$ and $\text{SAT}(\text{tw}_{\mathcal{C}})$ are fixed-parameter tractable.*

17.6.2. Modularity

Networks that arise from real-world applications frequently exhibit a certain *community structure*, where nodes form strongly interconnected communities which are sparsely connected with each other. With the notion of *modularity* [New03, New06, NG04, ZPWL13] one can measure to what extent a network exhibits such a structure. It was empirically observed, that the performance of SAT solvers shows some correlation with the modularity of the input formulas [ABGL14, NGF⁺14]. However, the presence of a community structure is not a guarantee for a formula to be tractable. In fact, it is not difficult to show that SAT remains NP-hard for highly modular instances [GS15, MFS16]. Based on this observation, Ganian and Szeider proposed the satisfiability parameter *h-modularity* which is inspired by the general concept of modularity, but in contrast to the existing notion does provide performance guarantees for SAT decision and even model counting. This satisfiability parameter is based on the splitting of the clauses of a given formula into classes, called *h-communities*, where each class is a hitting formula (see Section 17.4.4), and therefore strongly interconnected. The *h-communities* are only sparsely connected with each other, as the graph representing their interconnection (the *community graph*) as small treewidth.

More specifically, we call a subset H of a formula F to be a *hitting community* (or *h-community* in brief) in F if H is a hitting formula. The *degree* $\deg(H)$ of an *h-community* H is the number of edges in the dual graph of F between a clause in H and a clause outside of H . A *hitting community structure* (or *h-structure* in brief) \mathcal{P} is a partitioning of F into *h-communities*, and the degree $\deg(\mathcal{P})$ of \mathcal{P} is $\max\{\deg(H) : H \in \mathcal{P}\}$.

To measure the treewidth of an *h-structure* \mathcal{P} , we construct a *community graph* $G(\mathcal{P})$ as follows. The vertices of $G(\mathcal{P})$ are the *h-communities* in \mathcal{P} , and two vertices A, B in $G(\mathcal{P})$ are joined by an edge if and only if there exist clauses $C \in A$ and $D \in B$ which are adjacent.

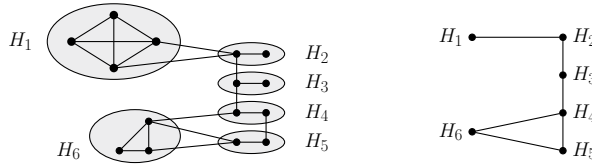


Figure 17.7. The dual graph (left) and community graph (right) of the formula F and the h-structure \mathcal{P} .

We define the *h-modularity* of an h-structure \mathcal{P} as the maximum over $\deg(\mathcal{P})$ and $\text{tw}(G(\mathcal{P}))$. The h-modularity $\mathbf{h-mod}(F)$ of a formula F is then defined as the minimum $\mathbf{h-mod}(\mathcal{P})$ over all h-structures \mathcal{P} of F .

Example 17.6.1. Consider the formula F which can be split into h-communities $H_1 = \{\{x, y, \neg a\}, \{\neg x, y, a\}, \{x, \neg y\}, \{\neg x, \neg y\}\}$, $H_2 = \{\{a, b, c\}, \{\neg b\}\}$, $H_3 = \{\{c, \neg d, e, f\}, \{d, \neg e\}\}$, $H_4 = \{\{f, \neg g, \neg h\}, \{h, \neg i\}\}$, $H_5 = \{\{i, \neg j\}, \{j, k, l, \neg m, \neg n\}\}$, and $H_6 = \{\{u, \neg v, g, \neg k, l, m, \neg n\}, \{u, v, \neg l, \neg u, v\}\}$. Figure 17.7 (left) shows the dual graph of F with the h-communities indicated. Figure 17.7 (right) shows the corresponding community graph, which is of treewidth 2. The h-communities H_1 and H_3 have degree 2, and all other h-communities have degree 3. Therefore the h-modularity of F is at most $\max(3, 2) = 3$. \square

Once we know a community structure \mathcal{P} of a CNF formula F , then we can solve **SAT** and **#SAT** by a fixed-parameter algorithm for parameter $\mathbf{h-mod}(\mathcal{P})$ by compiling it into an instance of the Sum-of-Products problems (the counting version of CSP), and solve it using known results [BDP09, GKSS18]. For actually finding a community structure \mathcal{P} of small h-modularity, we know a fixed-parameter approximation algorithm [GS15].

Theorem 17.6.3 ([GS15]). *The problem $\mathbf{\#SAT}(\mathbf{h-mod})$ is fixed-parameter tractable.*

17.6.3. Matchings

A *matching* in a graph is a set of edges such that every vertex is incident with at most one edge of the matching. A CNF formula is called *matched* if its incidence graph has a matching such that all clauses are incident with an edge of the matching. Matched formulas are always satisfiable since one can satisfy each clause independently by choosing the right truth value for the variable that is associated with it via the matching.

Example 17.6.2. Consider the CNF formula $F = \{C_1, \dots, C_4\}$ with $C_1 = \{v, y, z\}$, $C_2 = \{\bar{y}, \bar{x}\}$, $C_3 = \{\bar{v}, \bar{z}, w\}$, $C_4 = \{y, x, \bar{w}\}$. The set $M = \{vC_1, yC_2, zC_3, xC_4\}$ is a matching in the incidence graph of F that covers all clauses. Hence F is a matched formula and it is indeed satisfiable: we put $\tau(v) = 1$ to satisfy C_1 , $\tau(y) = 0$ to satisfy C_2 , $\tau(z) = 0$ to satisfy C_3 , and $\tau(x) = 1$ to satisfy C_4 . \square

The notion of *maximum deficiency* (first used by Franco and Van Gelder [FV03] in the context of CNF formulas) allows to gradually extend the nice properties from matched formulas to more general classes of formulas. The maximum deficiency of a formula F , denoted by $\mathbf{md}(F)$, is the number of clauses remaining uncovered by a largest matching of the incidence graph of F . The parameters \mathbf{md} and \mathbf{tw}^* are domination incomparable [Sze04b]. The term “maximum deficiency” is motivated by the equality

$$\mathbf{md}(F) = \max_{F' \subseteq F} \mathbf{d}(F')$$

which follows from Hall’s Theorem. Here $\mathbf{d}(F')$ denotes the *deficiency* of F' , the difference $|F'| - |\text{var}(F')|$ between the number of clauses and the number of variables. The problem $\mathbf{VER}(\mathbf{md})$ can be solved in polynomial time, since a largest matching in a bipartite graph can be found in polynomial time through Hopcroft and Karp’s algorithm [HK73, LP86] (and the number of uncovered clauses remains the same whatever largest matching one considers).

Deficiency and maximum deficiency have been studied in the context of *minimal unsatisfiable formulas*, i.e., unsatisfiable formulas that become satisfiable by removing any clause. Let \mathbf{MU} denote the recognition problem for minimal unsatisfiable formulas. By a classic result of Papadimitriou and Wolfe [PW88], the problem \mathbf{MU} is DP-complete; DP is the class of problems that can be considered as the difference of two problems in NP and corresponds to the second level of the Boolean Hierarchy [Joh90]. Kleine Büning [Kle00] initiated the study of \mathbf{MU} parameterized by the deficiency \mathbf{d} . Since $\mathbf{d}(F) = \mathbf{md}(F) \geq 1$ holds for minimal unsatisfiable formulas F [AL86], algorithms for $\mathbf{SAT}(\mathbf{md})$ are of relevance. Fleischer et al. [FKS02] have shown that one can decide the satisfiability of formulas with maximum deficiency bounded by a constant in polynomial time.

As a consequence, minimal unsatisfiable formulas with deficiency bounded by a constant can be recognized in polynomial time. The order of the polynomial that bounds the running time of Fleischer et al.’s algorithm depends on k ; hence, it only establishes that $\mathbf{SAT}(\mathbf{md})$ and $\mathbf{MU}(\mathbf{d})$ are in XP. Szeider [Sze04a] developed an algorithm that decides satisfiability and minimal unsatisfiability of formulas with maximum deficiency k in time $\mathcal{O}^*(2^k)$, thus establishing the following result.

Theorem 17.6.4 ([Sze04a]). *The problems $\mathbf{SAT}(\mathbf{md})$ and $\mathbf{MU}(\mathbf{d})$ are fixed-parameter tractable.*

Key for Szeider’s algorithm is a polynomial-time procedure that either decides the satisfiability of a given formula F or reduces F to an equisatisfiable formula F^* with $\mathbf{md}(F^*) \leq \mathbf{md}(F)$, such that

$$\mathbf{md}(F^*[x = 0]) < \mathbf{md}(F^*) \text{ and } \mathbf{md}(F^*[x = 1]) < \mathbf{md}(F^*) \text{ for all } x \in \text{var}(F^*);$$

a formula F^* with this property is called *\mathbf{md} -critical*. In particular, a formula is \mathbf{md} -critical if every literal of F^* occurs in at least two clauses and for every non-empty set X of variables of F^* there are at least $|X| + 2$ clauses C of F^* such that $\text{var}(C) \cap X \neq \emptyset$. The above reduction is applied at every node of a (DPLL-type) binary search tree. Since at every step from a node to one of its

children the maximum deficiency of the formula gets reduced, it follows that the height of the search tree is bounded in terms of the maximum deficiency of the given formula, yielding the fixed-parameter tractability of **SAT**(**md**).

Let r be a positive integer and let \mathcal{M}_r denote the class of formulas F with $\mathbf{md}(F) \leq r$. Since both recognition and satisfiability of formulas in \mathcal{M}_r can be solved in polynomial time and, since \mathcal{M}_r is clause induced, it makes sense to consider \mathcal{M}_r as the base class for strong and deletion backdoor sets. However, such backdoor sets are difficult to find:

Theorem 17.6.5 ([Sze08]). *The problems $\mathbf{VER}(\mathbf{b}_{\mathcal{M}_r})$ and $\mathbf{VER}(\mathbf{db}_{\mathcal{M}_r})$ are $W[2]$ -hard for every $r \geq 1$.*

17.7. Concluding Remarks

We close this chapter by briefly mentioning further research on the parameterized complexity of problems related to propositional satisfiability.

For example, Fellows, Szeider and Wrightson [FSW06] have studied the problem of finding in a given CNF formula F a *small unsatisfiable subset* S parameterized by the number of clauses of S . The problem is $W[1]$ -complete, but fixed-parameter tractable for several classes of CNF formulas, including formulas with planar incidence graphs and formulas with both clause size and occurrence of variables bounded. $W[1]$ -hardness prevails if the input formula is Horn [dHKS17]. This is in stark contrast to the case where the input is a 2CNF formula; then the problem can be solved even in polynomial time [BM07].

Propositional proof complexity is a further area of research that is related to satisfiability and admits parameterizations. In particular, one can study proofs that establish that a given CNF formula cannot be satisfied by setting at most k variables to true; k is considered as the parameter. Dantchev, Martin, and Szeider [DMS11] have studied the proof complexity of resolution for such “parameterized contradictions,” with very interesting follow-up work [ABdR⁺18, BGLR12].

Another promising line of research that connects parameterized complexity with SAT solving is the exploration of *fixed-parameter tractable reductions to SAT*. The idea is to solve problems that are believed to be harder than NP (e.g., problems from the second level of the Polynomial Hierarchy) by a fixed-parameter algorithm that can call a SAT solver as an auxiliary device. Such algorithms were developed for disjunctive answer-set programming and propositional abduction problems [PRS13, FS15]. De Haan and Szeider [dHS17, dHS19] developed a parameterized hardness theory for such problems which lies the grounds for classifying problems that admit such fixed-parameter tractable reductions to SAT, and those that don’t.

We hope that this survey provides a stimulating starting point for further research on satisfiability and related topics that fruitfully utilizes concepts of parameterized complexity theory.

Acknowledgment

This work was supported by the EPSRC, project EP/E001394/1 “Fixed-Parameter Algorithms and Satisfiability.”

References

- [ABdR⁺18] A. Atserias, I. Bonacina, S. F. de Rezende, M. Lauria, J. Nordström, and A. A. Razborov. Clique is hard on average for regular resolution. In *STOC*, pages 866–877. ACM, 2018.
- [ABGL14] C. Ansótegui, M. L. Bonet, J. Giráldez-Cru, and J. Levy. The fractal dimension of SAT formulas. In S. Demri, D. Kapur, and C. Weidenbach, editors, *Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings*, volume 8562 of *LNCS*, pages 107–121. Springer-Verlag, 2014.
- [ACP87] S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k -tree. *SIAM Journal on Algebraic and Discrete Methods*, 8(2):277–284, 1987.
- [AGK⁺11] N. Alon, G. Gutin, E. J. Kim, S. Szeider, and A. Yeo. Solving MAX-r-SAT above a tight lower bound. *Algorithmica*, 61(3):638–655, 2011.
- [AL86] R. Aharoni and N. Linial. Minimal non-two-colorable hypergraphs and minimal unsatisfiable formulas. *Journal of Combinatorial Theory, Series A*, 43(2):196–204, 1986.
- [BBE17] M. Bannach, S. Berndt, and T. Ehlers. Jdrasil: A modular library for computing tree decompositions. In C. S. Iliopoulos, S. P. Pissis, S. J. Puglisi, and R. Raman, editors, *16th International Symposium on Experimental Algorithms, SEA 2017, June 21-23, 2017, London, UK*, volume 75 of *LIPICs*, pages 28:1–28:21. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.
- [BCH90] E. Boros, Y. Crama, and P. L. Hammer. Polynomial-time inference of all valid implications for Horn and related formulae. *Ann. Math. Artif. Intell.*, 1:21–32, 1990.
- [BDD⁺16] H. L. Bodlaender, P. I. G. n. s. Drange, M. S. Dregi, F. V. Fomin, D. Lokshantov, and M. Pilipczuk. A $c^k n$ 5-approximation algorithm for treewidth. *SIAM Journal of Computing*, 45(2):317–378, 2016.
- [BDP03] F. Bacchus, S. Dalmao, and T. Pitassi. Algorithms and complexity results for #SAT and Bayesian inference. In *Proc. 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS'03)*, pages 340–351. IEEE Computer Society, 2003.
- [BDP09] F. Bacchus, S. Dalmao, and T. Pitassi. Solving #SAT and Bayesian inference with backtracking search. *J. Artif. Intell. Res.*, 34:391–442, 2009.
- [BFK⁺06] H. L. Bodlaender, F. V. Fomin, A. M. C. A. Koster, D. Kratsch, and D. M. Thilikos. On exact algorithms for treewidth. In *Proc. 14th Annual European Symposium on Algorithms (ESA'06)*, volume 4168 of *LNCS*, pages 672–683. Springer-Verlag, 2006.
- [BGLR12] O. Beyersdorff, N. Galesi, M. Lauria, and A. A. Razborov. Parameterized bounded-depth frege is not optimal. *ACM Trans. Comput. Theory*, 4(3):7:1–7:16, 2012.
- [BHKK07] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto. Fourier meets möbius: fast subset convolution. In *STOC*, pages 67–74. ACM, 2007.

- [BHS94] E. Boros, P. L. Hammer, and X. Sun. Recognition of q -Horn formulae in linear time. *Discrete Applied Mathematics*, 55(1):1–13, 1994.
- [BJ14] J. Berg and M. Järvisalo. SAT-based approaches to treewidth computation: An evaluation. In *Proceedings of the 26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI'14*, pages 328–335, Limassol, Cyprus, November 2014. IEEE Computer Soc.
- [Bjö18] A. Björklund. Personal Communication, 2018.
- [BK96] H. L. Bodlaender and T. Kloks. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *Journal of Algorithms*, 21(2):358–402, 1996.
- [BM07] J. Buresh-Oppenheim and D. G. Mitchell. Minimum 2CNF resolution refutations in polynomial time. In J. Marques-Silva and K. A. Sakallah, editors, *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings*, volume 4501 of *LNCS*, pages 300–313. Springer-Verlag, 2007.
- [Bod93] H. L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11(1-2):1–22, 1993.
- [Bod96] H. L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal of Computing*, 25(6):1305–1317, 1996.
- [Bod05] H. L. Bodlaender. Discovering treewidth. In *Proc. 31st Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'05)*, volume 3381 of *LNCS*, pages 1–16. Springer-Verlag, 2005.
- [BS17] S. Bova and S. Szeider. Circuit treewidth, sentential decision, and query compilation. In E. Sallinger, J. V. den Bussche, and F. Geerts, editors, *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 233–246. ACM, 2017.
- [BW04] F. Bacchus and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. In *Proc. 6th International Conference on Theory and Applications of Satisfiability Testing (SAT'03), Selected and Revised Papers*, volume 2919 of *LNCS*, pages 341–355. Springer-Verlag, 2004.
- [CCDF97] L. Cai, J. Chen, R. G. Downey, and M. R. Fellows. Advice classes of parameterized tractability. *Annals of Pure and Applied Logic*, 84(1):119–138, 1997.
- [CEH97] Y. Crama, O. Ekin, and P. L. Hammer. Variable and term removal from Boolean formulae. *Discrete Applied Mathematics*, 75(3):217–230, 1997.
- [CER93] B. Courcelle, J. Engelfriet, and G. Rozenberg. Handle-rewriting hypergraph grammars. *Journal of Computer and System Sciences*, 46(2):218–270, 1993.
- [Ces06] M. Cesati. Compendium of parameterized problems. <http://cesati.sprg.uniroma2.it/research/compendium/>, September 2006.

- [CFK⁺13] M. Cygan, F. V. Fomin, L. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized Algorithms*. Texts in Computer Science. Springer, 2013.
- [CJG08] D. Cohen, P. Jeavons, and M. Gyssens. A unified theory of structural tractability for constraint satisfaction problems. *Journal of Computer and System Sciences*, 74(5):721–743, 2008.
- [CKJ01] J. Chen, I. A. Kanj, and W. Jia. Vertex cover: Further observations and further improvements. *Journal of Algorithms*, 41(2):280–301, 2001.
- [CKX10] J. Chen, I. A. Kanj, and G. Xia. Improved upper bounds for vertex cover. *Theoretical Computer Science*, 411(40-42):3736–3756, 2010.
- [CMR01] B. Courcelle, J. A. Makowsky, and U. Rotics. On the fixed parameter complexity of graph enumeration problems definable in monadic second-order logic. *Discrete Applied Mathematics*, 108(1-2):23–52, 2001.
- [CO00] B. Courcelle and S. Olariu. Upper bounds to the clique-width of graphs. *Discrete Applied Mathematics*, 101(1-3):77–114, 2000.
- [Cou88] B. Courcelle. The monadic second-order logic of graphs: Definable sets of finite graphs. In *Proc. 14th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'88)*, volume 344 of *LNCS*, pages 30–53. Springer-Verlag, 1988.
- [CXW17] J. Chen, C. Xu, and J. Wang. Dealing with 4-variables by resolution: an improved MaxSAT algorithm. *Theoretical Computer Science*, 670:33–44, 2017.
- [DF99] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer-Verlag, 1999.
- [DF13] R. G. Downey and M. R. Fellows. *Fundamentals of Parameterized Complexity*. Texts in Computer Science. Springer, 2013.
- [DGS07] B. N. Dilkina, C. P. Gomes, and A. Sabharwal. Tradeoffs in the complexity of backdoor detection. In *Proc. 13th International Conference on Principles and Practice of Constraint Programming (CP'07)*, volume 4741 of *LNCS*, pages 256–270. Springer-Verlag, 2007.
- [dHKS17] R. de Haan, I. Kanj, and S. Szeider. On the parameterized complexity of finding small unsatisfiable subsets of CNF formulas and CSP instances. *ACM Trans. Comput. Log.*, 18(3):Art. 21, 46, 2017.
- [dHS17] R. de Haan and S. Szeider. Parameterized complexity classes beyond Para-NP. *Journal of Computer and System Sciences*, 87:16–57, 2017.
- [dHS19] R. de Haan and S. Szeider. Compendium of parameterized problems at higher levels of the polynomial hierarchy. *MDPI Algorithms*, 12(9):1–28, 2019.
- [DMS11] S. S. Dantchev, B. Martin, and S. Szeider. Parameterized proof complexity. *Computational Complexity*, 20(1):51–85, 2011.
- [Fel03] M. R. Fellows. Blow-ups, win/win's, and crown rules: Some new directions in fpt. In H. L. Bodlaender, editor, *Graph-Theoretic Concepts in Computer Science (WG 2003)*, volume 2880 of *LNCS*, pages 1–12. Springer-Verlag, 2003.
- [FG06] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Springer-

- Verlag, 2006.
- [FKS02] H. Fleischner, O. Kullmann, and S. Szeider. Polynomial-time recognition of minimal unsatisfiable formulas with fixed clause-variable difference. *Theoretical Computer Science*, 289(1):503–516, 2002.
- [FKT04] F. V. Fomin, D. Kratsch, and I. Todinca. Exact (exponential) algorithms for treewidth and minimum fill-in. In *Proc. 31st International Colloquium on Automata, Languages and Programming (ICALP'04)*, volume 3142 of *LNCS*, pages 568–580. Springer-Verlag, 2004.
- [FLM⁺15] F. V. Fomin, D. Lokshtanov, N. Misra, M. S. Ramanujan, and S. Saurabh. Solving d -sat via backdoors to small treewidth. In P. Indyk, editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 630–641. SIAM, 2015.
- [FLS17] J. K. Fichte, N. Lodha, and S. Szeider. SAT-based local improvement for finding tree decompositions of small width. In S. Gaspers and T. Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10491 of *LNCS*, pages 401–411. Springer-Verlag, 2017.
- [FMR08] E. Fischer, J. A. Makowsky, and E. V. Ravve. Counting truth assignments of formulas of bounded tree-width or clique-width. *Discrete Applied Mathematics*, 156(4):511–529, 2008. DOI: 10.1016/j.dam.2006.06.020.
- [FS15] J. K. Fichte and S. Szeider. Backdoors to tractable answer set programming. *Artificial Intelligence*, 220:64–103, March 2015.
- [FSW06] M. R. Fellows, S. Szeider, and G. Wrightson. On finding short resolution refutations and small unsatisfiable subsets. *Theoretical Computer Science*, 351(3):351–359, 2006.
- [FV03] J. Franco and A. Van Gelder. A perspective on certain polynomial time solvable classes of satisfiability. *Discrete Applied Mathematics*, 125(2):177–214, 2003.
- [GHO13] R. Ganian, P. Hlinený, and J. Obdržálek. Better algorithms for satisfiability problems for formulas of bounded rank-width. *Fund. Inform.*, 123(1):59–76, 2013.
- [GJ79] M. R. Garey and D. R. Johnson. *Computers and Intractability*. W. H. Freeman and Company, 1979.
- [GKSS18] R. Ganian, E. J. Kim, F. Slivovsky, and S. Szeider. Sum-of-products with default values: Algorithms and complexity results. In M. Alamaniotis, editor, *Proceedings of ICTAI 2018, the 30th IEEE International Conference on Tools with Artificial Intelligence*, 2018. To appear.
- [GLS02] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. *Journal of Computer and System Sciences*, 64(3):579–627, 2002.
- [GM06] M. Grohe and D. Marx. Constraint solving via fractional edge covers. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '06)*, pages 289–298. ACM Press, 2006.

- [GMO⁺17] S. Gaspers, N. Misra, S. Ordyniak, S. Szeider, and S. Zivny. Backdoors into heterogeneous classes of SAT and CSP. *Journal of Computer and System Sciences*, 85:38–56, 2017.
- [GN07] J. Guo and R. Niedermeier. Invitation to data reduction and problem kernelization. *ACM SIGACT News*, 38(2):31–45, 2007.
- [GOR⁺16] S. Gaspers, S. Ordyniak, M. S. Ramanujan, S. Saurabh, and S. Szeider. Backdoors to q-Horn. *Algorithmica*, 74(1):540–557, 2016.
- [GRS17a] R. Ganian, M. S. Ramanujan, and S. Szeider. Backdoor treewidth for SAT. In S. Gaspers and T. Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10491 of *LNCS*, pages 20–37. Springer-Verlag, 2017.
- [GRS17b] R. Ganian, M. S. Ramanujan, and S. Szeider. Combining Treewidth and Backdoors for CSP. In H. Vollmer and B. Vallee, editors, *34th Symposium on Theoretical Aspects of Computer Science (STACS 2017)*, volume 66 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 36:1–36:17, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- [GS08] G. Gottlob and S. Szeider. Fixed-parameter algorithms for artificial intelligence, constraint satisfaction, and database problems. *The Computer Journal*, 51(3):303–325, 2008. DOI: 10.1093/comjnl/bxm056.
- [GS12a] S. Gaspers and S. Szeider. Backdoors to acyclic SAT. In A. Czumaj, K. Mehlhorn, A. M. Pitts, and R. Wattenhofer, editors, *Automata, Languages, and Programming - 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Proceedings, Part I*, volume 7391 of *LNCS*, pages 363–374. Springer-Verlag, 2012.
- [GS12b] S. Gaspers and S. Szeider. Backdoors to satisfaction. In H. L. Bodlaender, R. Downey, F. V. Fomin, and D. Marx, editors, *The Multivariate Algorithmic Revolution and Beyond - Essays Dedicated to Michael R. Fellows on the Occasion of His 60th Birthday*, volume 7370 of *LNCS*, pages 287–317. Springer-Verlag, 2012.
- [GS13] S. Gaspers and S. Szeider. Strong backdoors to bounded treewidth SAT. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, pages 489–498. IEEE Computer Society, 2013.
- [GS14] S. Gaspers and S. Szeider. Guarantees and limits of preprocessing in constraint satisfaction and reasoning. *Artificial Intelligence*, 216:1–19, 2014.
- [GS15] R. Ganian and S. Szeider. Community structure inspired algorithms for SAT and #SAT. In M. Heule and S. Weaver, editors, *18th International Conference on Theory and Applications of Satisfiability Testing (SAT 2015), September 24-27, 2015, Austin, Texas*, number 9340 in *LNCS*, pages 223–237. Springer-Verlag, 2015.
- [GS17] R. Ganian and S. Szeider. New width parameters for model counting. In S. Gaspers and T. Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference*,

- Melbourne, VIC, Australia, August 28 - September 1, 2017, *Proceedings*, volume 10491 of *LNCS*, pages 38–52. Springer-Verlag, 2017.
- [GSS02] G. Gottlob, F. Scarcello, and M. Sideri. Fixed-parameter complexity in AI and nonmonotonic reasoning. *Artificial Intelligence*, 138(1-2):55–86, 2002.
- [HK73] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal of Computing*, 2(4):225–231, 1973.
- [HS15] M. Heule and S. Szeider. A SAT approach to clique-width. *ACM Trans. Comput. Log.*, 16(3):24, 2015.
- [IPZ01] R. Impagliazzo, R. Paturi, and F. Zane. Which problems have strongly exponential complexity. *Journal of Computer and System Sciences*, 63(4):512–530, 2001.
- [Iwa89] K. Iwama. CNF-satisfiability test by counting and polynomial average time. *SIAM Journal of Computing*, 18(2):385–391, 1989.
- [Joh73] D. S. Johnson. Approximation algorithms for combinatorial problems. In *Fifth Annual ACM Symposium on Theory of Computing (Austin, Tex., 1973)*, pages 38–49. Assoc. Comput. Mach., New York, 1973.
- [Joh90] D. S. Johnson. A catalog of complexity classes. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 2, pages 67–161. Elsevier Science Publishers, 1990.
- [JS12] A. K. Jha and D. Suciu. On the tractability of query compilation and bounded treewidth. In A. Deutsch, editor, *15th International Conference on Database Theory, ICDT '12, Berlin, Germany, March 26-29, 2012*, pages 249–261. ACM, 2012.
- [KBL99] H. Kleine Büning and T. Lettmann. *Propositional logic: Deduction and algorithms*. Cambridge University Press, 1999.
- [KBvH01a] A. M. C. A. Koster, H. L. Bodlaender, and S. P. M. van Hoesel. Treewidth: Computational experiments. Technical Report ZIB 01-38, Konrad-Zuse-Zentrum für Informationstechnik Berlin, 2001.
- [KBvH01b] A. M. C. A. Koster, H. L. Bodlaender, and S. P. M. van Hoesel. Treewidth: Computational experiments. *Electronic Notes in Discrete Mathematics*, 8:54–57, 2001.
- [Kle00] H. Kleine Büning. On subclasses of minimal unsatisfiable formulas. *Discrete Applied Mathematics*, 107(1-3):83–98, 2000.
- [Klo94] T. Kloks. *Treewidth: Computations and Approximations*. Springer-Verlag, 1994.
- [KV00] P. G. Kolaitis and M. Y. Vardi. Conjunctive-query containment and constraint satisfaction. *Journal of Computer and System Sciences*, 61(2):302–332, 2000.
- [KZ01] H. Kleine Büning and X. Zhao. Satisfiable formulas closed under replacement. *Electronic Notes in Discrete Mathematics*, 9:48–58, 2001.
- [Lew78] H. R. Lewis. Renaming a set of clauses as a Horn set. *Journal of the ACM*, 25(1):134–135, 1978.
- [LP86] L. Lovász and M. D. Plummer. *Matching Theory*. Number 29 in *Annals of Discrete Mathematics*. North-Holland Publishing Co., Ams-

- terdam, 1986.
- [MFS16] N. Mull, D. J. Fremont, and S. A. Seshia. On the hardness of SAT with community structure. In *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *LNCS*, pages 141–159. Springer, 2016.
- [MR99] M. Mahajan and V. Raman. Parameterizing above guaranteed values: MaxSat and MaxCut. *Journal of Algorithms*, 31(2):335–354, 1999.
- [MRS06] M. Mahajan, V. Raman, and S. Sikdar. Parameterizing MAX SNP problems above guaranteed values. In *Proc. 2nd International Workshop on Parameterized and Exact Computation (IWPEC'06)*, volume 4169 of *LNCS*, pages 38–49. Springer-Verlag, 2006.
- [MS10] D. Marx and I. Schlotter. Parameterized complexity and local search approaches for the stable marriage problem with ties. *Algorithmica*, 58(1):170–187, 2010.
- [MS11] D. Marx and I. Schlotter. Stable assignment with couples: parameterized complexity and local search. *Discrete Optim.*, 8(1):25–40, 2011.
- [New03] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003.
- [New06] M. E. J. Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 103(23):8577–8582, 2006.
- [NG04] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Phys. Rev. E*, 69(2):026113, February 2004.
- [NGF⁺14] Z. Newsham, V. Ganesh, S. Fischmeister, G. Audemard, and L. Simon. Impact of community structure on SAT solver performance. In C. Sinz and U. Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *LNCS*, pages 252–268. Springer-Verlag, 2014.
- [Nie06] R. Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Number 31 in Oxford Lecture Series in Mathematics and Its Applications. Oxford University Press, 2006.
- [NRS04] N. Nishimura, P. Ragde, and S. Szeider. Detecting backdoor sets with respect to Horn and binary clauses. In *Proc. 7th International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, pages 96–103. Informal Proceedings, 2004.
- [NRS07] N. Nishimura, P. Ragde, and S. Szeider. Solving #SAT using vertex covers. *Acta Informatica*, 44(7-8):509–523, 2007.
- [OPS13] S. Ordyniak, D. Paulusma, and S. Szeider. Satisfiability of acyclic and almost acyclic CNF formulas. *Theoretical Computer Science*, 481:85–99, 2013.
- [OSS21] S. Ordyniak, A. Schidler, and S. Szeider. Backdoor DNFs. Technical Report AC-TR-21, Algorithms and Complexity Group, TU Wien,

- 2021.
- [Oum05] S.-i. Oum. Approximating rank-width and clique-width quickly. In *Proc. 31st International Workshop on Graph-Theoretic Concepts in Computer Science (WG'05)*, volume 3787 of *LNCS*, pages 49–58. Springer-Verlag, 2005.
- [Par16] A. Parlak. A SAT approach to clique-width of a digraph and an application on model counting problem. Master's thesis, TU Wien, Algorithms and Complexity Group, 2016. Supervised by Stefan Szeider.
- [PRS13] A. Pfandler, S. Rümmele, and S. Szeider. Backdoors to abduction. In *Proceedings of IJCAI 2013, the 23th International Joint Conference on Artificial Intelligence, August 3-9, 2013, Beijing, China*, 2013.
- [PSS16] D. Paulusma, F. Slivovsky, and S. Szeider. Model counting for CNF formulas of bounded modular treewidth. *Algorithmica*, 76(1):168–194, 2016.
- [PW88] C. H. Papadimitriou and D. Wolfe. The complexity of facets resolved. *Journal of Computer and System Sciences*, 37(1):2–13, 1988.
- [Ree92] B. A. Reed. Finding approximate separators and computing tree-width quickly. In *Proc. 24th Annual ACM symposium on Theory of Computing (STOC'92)*, pages 221–228. ACM Press, 1992.
- [RO08] I. Razgon and B. O'Sullivan. Almost 2-SAT is fixed-parameter tractable. In *Proc. 35th International Colloquium on Automata, Languages and Programming (ICALP'08), Track A: Algorithms, Automata, Complexity, and Games*, volume 5125 of *LNCS*, pages 551–562. Springer-Verlag, 2008.
- [Rot96] D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1-2):273–302, 1996.
- [RS86] N. Robertson and P. D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, 1986.
- [RS91] N. Robertson and P. D. Seymour. Graph minors X. Obstructions to tree-decomposition. *Journal of Combinatorial Theory, Series B*, 52(2):153–190, 1991.
- [RS17] M. S. Ramanujan and S. Saurabh. Linear-time parameterized algorithms via skew-symmetric multicuts. *ACM Transactions on Algorithms*, 13(4):Art. 46, 25, 2017.
- [RTL76] D. J. Rose, R. E. Tarjan, and G. S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal of Computing*, 5(2):266–283, 1976.
- [Sch78] T. J. Schaefer. The complexity of satisfiability problems. In *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing (San Diego, Calif., 1978)*, pages 216–226. ACM, 1978.
- [See91] D. Seese. The structure of the models of decidable monadic theories of graphs. *Annals of Pure and Applied Logic*, 53(2):169–195, 1991.
- [Spi03] J. P. Spinrad. *Efficient Graph Representations*. Fields Institute Monographs. AMS, 2003.
- [SS08] M. Samer and S. Szeider. Backdoor trees. In *Proc. 23rd AAAI Conference on Artificial Intelligence (AAAI'08)*, pages 363–368. AAAI

- Press, 2008.
- [SS09] M. Samer and S. Szeider. Backdoor sets of quantified Boolean formulas. *Journal of Automated Reasoning*, 42(1):77–97, 2009.
- [SS10a] M. Samer and S. Szeider. Algorithms for propositional model counting. *Journal of Discrete Algorithms*, 8(1):50–64, 2010.
- [SS10b] M. Samer and S. Szeider. Constraint satisfaction with bounded treewidth revisited. *Journal of Computer and System Sciences*, 76(2):103–114, 2010.
- [SS13] F. Slivovsky and S. Szeider. Model counting for formulas of bounded clique-width. In L. Cai, S. Cheng, and T. W. Lam, editors, *Algorithms and Computation - 24th International Symposium, ISAAC 2013, Hong Kong, China, December 16-18, 2013, Proceedings*, volume 8283 of *LNCS*, pages 677–687. Springer-Verlag, 2013.
- [SS20] F. Slivovsky and S. Szeider. A faster algorithm for propositional model counting parameterized by incidence treewidth. In L. Pulina and M. Seidl, editors, *Proceedings of SAT 2020, The 23rd International Conference on Theory and Applications of Satisfiability Testing*, volume 12178 of *LNCS*, pages 267–276. Springer-Verlag, 2020.
- [STV15] S. H. Sæther, J. A. Telle, and M. Vatshelle. Solving #sat and MAXSAT by dynamic programming. *J. Artif. Intell. Res.*, 54:59–82, 2015.
- [SV09] M. Samer and H. Veith. Encoding treewidth into SAT. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing, SAT 2009*, volume 5584 of *LNCS*, pages 45–50. Springer-Verlag, 2009.
- [Sze04a] S. Szeider. Minimal unsatisfiable formulas with bounded clause-variable difference are fixed-parameter tractable. *Journal of Computer and System Sciences*, 69(4):656–674, 2004.
- [Sze04b] S. Szeider. On fixed-parameter tractable parameterizations of SAT. In *Proc. 6th International Conference on Theory and Applications of Satisfiability Testing (SAT’03), Selected and Revised Papers*, volume 2919 of *LNCS*, pages 188–202. Springer-Verlag, 2004.
- [Sze05] S. Szeider. Backdoor sets for DLL subsolvers. *Journal of Automated Reasoning*, 35(1-3):73–88, 2005. Reprinted as Chapter 4 of the book “SAT 2005 – Satisfiability Research in the Year 2005”, edited by E. Giunchiglia and T. Walsh, Springer-Verlag, 2006.
- [Sze08] S. Szeider. Matched formulas and backdoor sets. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:1–12, 2008.
- [TY84] R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal of Computing*, 13(3):566–579, 1984.
- [Vil06] Y. Villanger. Improved exponential-time algorithms for treewidth and minimum fill-in. In *Proc. 7th Latin American Symposium on Theoretical Informatics (LATIN’06)*, volume 3887 of *LNCS*, pages 800–811. Springer-Verlag, 2006.
- [Wah17] M. Wahlström. *Algorithms, measures and upper bounds for satis-*

- fiability and related problems.* PhD thesis, Linköpings Universitet, 2017.
- [WGS03] R. Williams, C. P. Gomes, and B. Selman. On the connections between backdoors, restarts, and heavy-tailedness in combinatorial search. In *Proc. 6th International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, pages 222–230. Informal Proceedings, 2003.
- [Woe03] G. J. Woeginger. Exact algorithms for NP-hard problems: A survey. In *Proc. 5th International Workshop on Combinatorial Optimization (AUSSOIS'01) — “Eureka, You Shrink!”*, Revised Papers, volume 2570 of *LNCS*, pages 185–208. Springer-Verlag, 2003.
- [Yan81] M. Yannakakis. Algorithms for acyclic database schemes. In *Proc. 7th International Conference on Very Large Data Bases (VLDB'81)*, pages 81–94. IEEE Computer Society, 1981.
- [ZPWL13] W. Zhang, G. Pan, Z. Wu, and S. Li. Online community detection for large complex networks. In F. Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*. IJCAI/AAAI, 2013.

Part II

Applications and Extensions

This page intentionally left blank

Chapter 18

Bounded Model Checking

Armin Biere

Besides Equivalence Checking [KK97, KPKG02] the most important industrial application of SAT is currently Bounded Model Checking (BMC) [BCCZ99]. Both techniques are used for *formal* hardware verification in the context of electronic design automation (EDA), but have successfully been applied to many other domains as well. In this chapter, we focus on BMC.

In practice, BMC is mainly used for falsification resp. testing, which is concerned with violations of temporal properties. However, the original paper on BMC [BCCZ99] already discussed extensions that can prove properties. A considerable part of this chapter discusses these complete extensions, which are often called “unbounded” model checking techniques, even though they are build upon the same principles as plain BMC.

Two further related applications, in which BMC becomes more and more important, are automatic test case generation for closing coverage holes, and disproving redundancy in designs. Most of the techniques discussed in this chapter transfer to this more general setting as well, even though our focus is on property verification resp. falsification.

The basic idea of BMC is to represent a counterexample-trace of bounded length symbolically and check the resulting propositional formula with a SAT solver. If the formula is satisfiable and thus the path feasible, a satisfying assignment returned by the SAT solver can be translated into a concrete counterexample trace that shows that the property is violated. Otherwise, the bound is increased and the process repeated. Complete extensions to BMC allow to stop this process at one point, with the conclusion that the property cannot be violated, hopefully before the available resources are exhausted.

18.1. Model Checking

The origins of model checking go back to the seminal papers [CE82] and [QS82]. Clarke, Emerson and Sifakis won the 2007 Turing Award for their pioneering work on model checking. A workshop affiliated to the Federated Conference on Logic in Computer Science (FLOC’06) celebrated the 25th anniversary of model checking. The proceedings [VG08] of this workshop and *the* model checking book

[CGP99] are good starting points to learn about model checking. A more recent survey [PBG05] adds a perspective on SAT-based model checking.

In this chapter, we focus on SAT-based symbolic model checking [McM93], which originally relied on binary decision diagrams (BDDs) [Bry86] to symbolically represent systems. Operations on system states can then be mapped to BDD operations. In practice, BDDs can handle circuits with hundreds of latches, but often blow up in space.

BMC [BCCZ99] was an attempt to replace BDDs with SAT in symbolic model checking. However, SAT lacks the possibility to eliminate variables, which is a key operation in BDD-based model checking. The solution in BMC is to focus on falsification and, at least in a first approximation, drop completeness. This paradigm shift was hard to convey originally, but was accepted at the end, since SAT-based model checking, at least for falsification, scales much better [Kur08].

Another important direction in model checking is explicit state model checking. The SPIN model checker [Hol04] is the most prominent explicit state model checker and is mainly used for checking protocols. It draws its main power from partial order reduction techniques such as [Pel94]. Related techniques exist in BMC as well, see for instance [Hel01, JHN03]. However, for the rest of this chapter we focus on symbolic techniques for synchronous systems, for which partial order techniques do not seem to apply.

The first decade¹ of research in model checking witnessed a heated debate on which specification formalism is more appropriate: linear time logic versus branching time logic. Commonly only computation tree logic (CTL) [CE82], a branching time logic, and linear time logic (LTL) [Pnu77] are used. Originally, LTL was called propositional temporal logic (PTL) as a special case of “full” first-order temporal logic. However, nowadays LTL without further qualification is solely used for propositional linear temporal logic. Also note that PTL is also an acronym for past time (propositional) linear temporal, see for instance [BHJ⁺06].

LTL is arguably easier to understand and use, but at least in theory, LTL is harder [SC85] to check than CTL. If the system is represented symbolically, there is actually no difference as both problems are PSPACE complete [SC85, Sav70, PBG05]. Specifications in practice are typically in the intersection [Mai00] between LTL and CTL. If we restrict ourself to properties in the intersection, the problem of choosing between LTL and CTL boils down to which model checking algorithm to use. In this respect, BDD-based model checking has a slight bias towards CTL, whereas SAT-based model checking has a bias towards LTL. Thus, we mainly focus on LTL in the rest of this chapter. Further details on temporal logic and its history can be found in [Eme90, Var08].

18.1.1. LTL

As first promoted by Pnueli [Pnu77], temporal logic is an adequate specification formalism for concurrent resp. reactive systems. The syntax of the linear temporal logic LTL contains propositional boolean variables V , temporal operators and the usual propositional operators, including negation \neg and conjunction \wedge . Typical

¹A similar discussion took place in the recent process of standardizing temporal logic in the form of System Verilog Assertions (SVA) and the Property Specification Logic (PSL).

temporal operators are the “next time” operator **X**, the “finally” operator **F**, and the “globally” operator **G**.

Examples of temporal formulas are as follows: $a \rightarrow \mathbf{X}b$ means the property b has to hold in the next time instance, unless a does not hold now. With **X** alone only properties about a finite future around the initial state can be specified. The other temporal operators allow to extend this finite view, and specify infinite behavior. The “globally” operator **G** allows to specify *safety properties* in form of invariants or assertions that need to hold in all reachable states. For instance, $\mathbf{G}\neg(a \wedge b)$ specifies mutual exclusion of a and b . The only *liveness* operator we consider, the “finally” operator, describes necessary behavior, e.g. $\mathbf{G}(a \rightarrow \mathbf{F}b)$, which requires each a to be followed by b . More verbosely, the following invariant holds: if a is true then at the same time or later b has to hold, i.e. b cannot be postponed forever, after a has been assured. This is an invariant with a (potentially) liveness condition attached.

The interpretation of propositional variables may change over time but is uniquely determined by the current state of the model. This correspondence is captured via a labelling function $L: S \rightarrow \mathbb{P}(V)$, where S is the set of states. A propositional variable p is true in a system state s iff $p \in L(s)$. Beside the set of states S , a model has a set $I \subseteq S$ of initial states, and a transition relation $T \subseteq S \times S$. Such a model is also called *Kripke structure*. Often only models isomorphic to the set of interpretations of the boolean variables V are considered: then $S = \mathbb{P}(V)$ and $L(V') = V'$ for all “states” $s = V' \subseteq V$. A typical example are models of synchronous circuits, where V is the set of latches and input signals, and optionally includes output and internal signals as well. In the following, we fix one Kripke structure $K = (S, I, T, L)$ over the variables V .

The transition relation T is assumed to be total and the set I of initial states to be nonempty. As in the previous example, the transition relation is in general represented symbolically, e.g. as a circuit or a formula. In the following we simply write $T(s, s')$ for this formula, with the interpretation that $T(s, s')$ holds iff there is a transition from s to s' , also written as $s \rightarrow s'$. Note that s is simply a vector of all variables V in the current state and s' a vector of their primed copies in the successor state. We use a similar interpretation for $I(s)$.

The semantics of LTL are defined along paths of the model. A path π is an infinite sequence of states $\pi = (s_0, s_1, s_2, \dots)$, with $s_i \rightarrow s_{i+1}$. A path π is initialized if its first state $\pi(0) = s_0$ is an initial state. In the following, we also use the same notation to refer to single states of a path, e.g. $\pi(i) = s_i$ with $i \in \mathbb{N} = \{0, 1, 2, \dots\}$. A suffix of a path is defined as $\pi^i = (s_i, s_{i+1}, \dots)$. We now give a simplified² version of the standard (unbounded) semantics, defined recursively over the formula structure. An LTL formula f holds along a path π , written $\pi \models f$, iff

$$\begin{array}{llll} \pi \models p & \text{iff } p \in L(\pi(0)) & \pi \models \neg p & \text{iff } p \notin L(\pi(0)) \\ \pi \models g \vee h & \text{iff } \pi \models g \text{ or } \pi \models h & \pi \models g \wedge h & \text{iff } \pi \models g \text{ and } \pi \models h \\ \pi \models \mathbf{F}g & \text{iff } \exists j \in \mathbb{N}: \pi^j \models g & \pi \models \mathbf{G}g & \text{iff } \forall j \in \mathbb{N}: \pi^j \models g \\ \pi \models \mathbf{X}g & \text{iff } \pi^1 \models g & & \end{array}$$

²In particular we do not treat the “until” operator to make the following encoding easier to explain. The full semantics and its encoding can be found in [BHJ+06].

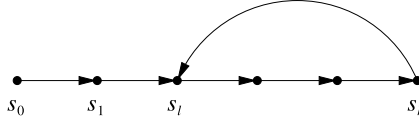


Figure 18.1. A (k, l) -lasso with $k = 5, l = 2$.

Here we assume that the formula is in negation normal form (NNF), i.e. negations are pushed down to the variables, with the help of the following axioms of LTL:

$$\neg(g \wedge h) \equiv (\neg g) \vee (\neg h) \quad \neg \mathbf{F}g \equiv \mathbf{G}\neg g \quad \neg \mathbf{G}g \equiv \mathbf{F}\neg g \quad \neg \mathbf{X}g \equiv \mathbf{X}\neg g$$

Finally, a formula f holds in a Kripke structure K , written $K \models f$, iff $\pi \models f$ for all initialized paths π of K . The model checking problem is to determine whether $K \models f$ holds. Related to the model checking problem is the question of the existence of a witness: a formula f has a witness in K iff there is an initialized path π with $\pi \models f$. Clearly $K \models f$ iff $\neg f$ does not have a witness in K . Therefore, we can reduce the model checking problem to the search for witnesses using negation and translation into NNF.

18.2. Bounded Semantics

First observe that some infinite paths can be represented by a finite prefix with a loop: an infinite path π is a (k, l) -lasso, iff $\pi(k + 1 + j) = \pi(l + j)$, for all $j \in \mathbb{N}$. In this case, π can actually be represented as $\pi = \pi_{\text{stem}} \cdot (\pi_{\text{loop}})^\omega$, as shown in Fig. 18.1.

As LTL enjoys a small model property [LP85], the search for witnesses can be restricted to lassos, if K is finite. See [BCCZ99, CKOS05] for more details. Let us rephrase the unbounded semantics by fixing the path π , but working with different suffixes π^i of π .

$$\begin{aligned} \pi^i \models p & \quad \text{iff} \quad p \in L(\pi(i)) & \quad \pi^i \models \neg p & \quad \text{iff} \quad p \notin L(\pi(i)) \\ \pi^i \models g \vee h & \quad \text{iff} \quad \pi^i \models g \text{ or } \pi^i \models h & \quad \pi^i \models g \wedge h & \quad \text{iff} \quad \pi^i \models g \text{ and } \pi^i \models h \\ \pi^i \models \mathbf{F}g & \quad \text{iff} \quad \exists j \in \mathbb{N}: \pi^{i+j} \models g & \quad \pi^i \models \mathbf{G}g & \quad \text{iff} \quad \forall j \in \mathbb{N}: \pi^{i+j} \models g \\ \pi^i \models \mathbf{X}g & \quad \text{iff} \quad \pi^{i+1} \models g \end{aligned}$$

To obtain “bounded semantics” we only look at the first $k + 1$ states and let i range over $0 \dots k$. If π is a (k, l) -lasso then $\pi^{k+1+j} = \pi^{l+j}$ for all $j \in \mathbb{N}$ and we get the following “bounded semantics” for lassos:

$$\begin{aligned} \pi^i \models \mathbf{F}g & \quad \text{iff} \quad \exists j \in \{\min(i, l), \dots, k\}: \pi^j \models g \\ \pi^i \models \mathbf{G}g & \quad \text{iff} \quad \forall j \in \{\min(i, l), \dots, k\}: \pi^j \models g \\ \pi^i \models \mathbf{X}g & \quad \text{iff} \quad \begin{cases} \pi^{i+1} \models g & \text{if } i < k \\ \pi^l \models g & \text{if } i = k \end{cases} \end{aligned}$$

For $\mathbf{G}g$ to hold on π^i , the body g has to hold at the current position i and of course at all larger positions j , with $i \leq j \leq k$. However, if the current position

i is in the loop, then g also has to hold from the loop start up to i . Using the minimum over the current position i and the loop start l covers both cases, no matter whether i is in the loop or still in the stem of π . A similar argument applies to $\mathbf{F}g$.

Now assume that π is *not* a (k, l) -lasso for any l . Then the suffix π^{k+1} of π can have an arbitrary shape. Looking only at the first $k + 1$ states of π is in general not enough to determine whether a formula holds along π , e.g. the “bounded semantics” can only be an approximation. Still the following but only sufficient conditions hold:

$$\begin{aligned} \pi^i \models \mathbf{F}g & \quad \text{if } \exists j \in [i \dots k]: \pi^j \models g \\ \pi^i \models \mathbf{X}g & \quad \text{if } \pi^{i+1} \models g \text{ and } i < k \end{aligned}$$

These early termination criteria are useful for providing counterexamples for pure safety formulas or to more general specifications with a safety part. For instance, in order to falsify the safety property $\mathbf{G}p$, we need to find a witness for $\mathbf{F}\neg p$. If p does not hold in some initial state s in K , then $k = 0$ is sufficient. All paths starting from s are actually witnesses, even if none of them is a $(0, 0)$ -loop.

If π is not a (k, l) -lasso for any l and we do not want to examine the suffix beyond the bound k , then we cannot conclude anything about $\pi^k \models \mathbf{X}g$ nor $\pi^i \models \mathbf{G}g$ for any $i < k$. This conservative approximation avoids reporting spurious witnesses. In propositional encodings of potential witnesses, we have to replace such LTL formulas by \perp , where \perp (\top) represents the boolean constant *false* (*true*).

18.3. Propositional Encodings

The bounded approximations of LTL semantics discussed above consider only the first $k + 1$ states of π . This is the key to obtain a propositional encoding of the LTL witness problem into SAT.

Assume that we have a symbolic representation of K and let s_0, \dots, s_k be vectors of copies of the state variables, i.e. for each time frame i , there is one copy V_i of V . Further let p_i denote the copy of p in time frame i . All encodings of the LTL witness problem include model constraints:

$$I(s_0) \wedge T(s_0, s_1) \wedge \dots \wedge T(s_{k-1}, s_k)$$

Looping constraints for $l \in \{0, \dots, k\}$ are added:

$$\lambda_l \rightarrow T(s_k, s_l)$$

We further assume that “at most” one λ_l holds. This cardinality constraint can be encoded with a circuit linear in k , for instance via Tseitin transformation [Tse68] of a BDD for this function.³

These constraints are always assumed. In particular, if the propositional encoding is satisfiable, the satisfying assignment can be interpreted as a prefix of

³Note that cardinality constraints and particularly at most constraints as in this case are symmetric functions, for which the variable order of the BDD does not matter: all reduced ordered BDDs (ROBDDs) for a specific symmetric function are isomorphic.

an initialized path π of K . If λ_l is assigned to \top then π is a (k, l) -loop. What remains to be encoded are the semantics of LTL, in order to make sure that π extracted from a satisfying assignment is indeed a witness.

18.3.1. Original Encoding

The original encoding [BCCZ99] of the witness problem of LTL into SAT, is a straightforward encoding of the reformulations resp. the bounded semantics. It can be implemented as a recursive procedure that takes as parameters an LTL formula f , a fixed bound k , the loop start l , and the position i . The last two parameters range between 0 and k . Let us denote with ${}_i[f]_k^i$ the resulting propositional formula obtained by encoding f for these parameters. First we make sure, by enforcing the model and looping constraints, that the symbolically represented path is a (k, l) -loop:

$$\begin{aligned} {}_i[p]_k^i &\equiv p_i & {}_i[\neg p]_k^i &\equiv \neg p_i \\ {}_i[g \vee h]_k^i &\equiv {}_i[g]_k^i \vee {}_i[h]_k^i & {}_i[g \wedge h]_k^i &\equiv {}_i[g]_k^i \wedge {}_i[h]_k^i \\ {}_i[\mathbf{F}g]_k^i &\equiv \bigvee_{j=\min(l,i)}^k {}_i[g]_k^j & {}_i[\mathbf{G}g]_k^i &\equiv \bigwedge_{j=\min(l,i)}^k {}_i[g]_k^j \\ {}_i[\mathbf{X}g]_k^i &\equiv {}_i[g]_k^j \text{ with } j = i + 1 \text{ if } i < k \text{ else } j = l \end{aligned}$$

Encoding witnesses without loops is similar. Let $[f]_k^i$ denote the result of encoding a witness without assuming that it is a (k, l) -loop for some l :

$$[\mathbf{F}g]_k^i \equiv \bigvee_{j=i}^k [g]_k^j \quad [\mathbf{G}g]_k^i \equiv \perp \quad [\mathbf{X}g]_k^i \equiv \begin{cases} [g]_k^{i+1} & \text{if } i < k \\ \perp & \text{if } j = k \end{cases}$$

The other cases are identical to the looping case. The full encoding is as follows:

$$[f]_k \equiv [f]_k^0 \vee \bigvee_{l=0}^k \lambda_l \wedge {}_l[f]_k^0$$

The second part handles (k, l) -loops, while the first part makes no assumption whether such a loop exists. With an inductive argument $[f]_k^0 \Rightarrow {}_l[f]_k^0$ follows. Therefore, there is no need to guard the left part with $\bigvee_{l=0}^k \lambda_l$, as it was originally presented in [BCCZ99].

For fixed k , there are $\Omega(|f| \cdot k^2)$ possible different parameters to ${}_i[f]_k^i$. In addition, an application of an equation introduces $\Omega(k)$ connectives to combine sub-results obtained from recursive calls. Thus, the overall complexity is at least cubic in k and linear in the size of the LTL formula $|f|$. For large k , this is not acceptable.⁴

⁴If the result of the encoding is represented as a circuit, then subformulas in the original encoding can be shared after restructuring the formula. In some cases this may even lead to a linear *circuit* encoding. But for binary temporal operators, including the “until” operator, this is in general not possible anymore.

18.3.2. Linear Encoding

The original encoding of [BCCZ99] presented in the last section is only efficient for simple properties such as $\mathbf{F}p$ or $\mathbf{G}p$, where $p \in V$. More involved specifications with deeply nested temporal operators produce quite some overhead. This is even more problematic for deeply nested binary temporal operators, such as the “until” operator \mathbf{U} , which we do not discuss in this chapter.

In symbolic model checking with BDDs, LTL is usually handled by a variant of the tableau construction of [LP85]. The tableau of [LP85] can be interpreted as a generalized Büchi automaton. It is conjuncted with K and a witness becomes a fair path on which all the fairness constraints occur infinitely often. In the context of Büchi automata, a fairness constraint is a set of states, which has to be “hit” by a path infinitely often, in order to be a fair path.

The LTL formula $\mathbf{GF}p$, i.e. infinitely often p , is a generic single fairness constraint with the following optimized encoding:

$$[\mathbf{GF}p]_k \equiv \bigvee_{l=0}^k \left(\lambda_l \wedge \bigvee_{i=l}^k p_i \right)$$

The formula is quadratic in k . However, it is possible to share common subformulas between different loop starts l . The resulting circuit is linear in k . This linear encoding of fairness constraints can be extended to multiple fairness constraints easily. Our first implementation of a bounded model checker used this technique in order to handle hundreds of fairness constraints [BCCZ99].

The tableau construction has symbolic variants [BCM⁺92, CGH97] as well as explicit variants [WVS83, VW94]. An explicit construction may explode in space immediately, since the Büchi automaton can be exponentially large in the size of the original LTL formula. This is rather unfortunate for BMC, but see [CKOS05] for a discussion on advantages and disadvantages of using an explicit automaton construction for BMC.

However, also symbolic tableau constructions – even with the presented optimized fairness encoding – require witnesses to be (k, l) -loops. This may prohibit early termination and requires larger bounds than necessary.

An improved but still quadratic encoding can be found in [FSW02]. A simpler and linear encoding was presented in [LBHJ04], which we explain next. A survey on the problem of encoding LTL (including past time LTL) can be found in [BHJ⁺06]. Another advanced encoding for weak alternating Büchi automata was presented in [HJK⁺06]. This encoding allows to handle all ω -regular properties which is a super set of LTL. All these symbolic encodings avoid the exponential blow-up of an explicit tableau construction and also allow to terminate earlier.

Before we explain the linear encoding, we first give an example why a simple recursive formulation is incorrect. Assume we have a single variable p and the following symbolic representation of a Kripke structure:

$$I(s) \equiv \bar{p} \qquad T(s, s') \equiv (s' = s)$$

The state space is $S = \{\perp, \top\}$, e.g. consists of all the valuations of p , see also Fig. 18.2. Only one state is reachable in K on which p does not hold. The

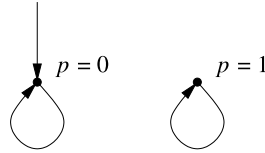


Figure 18.2. Kripke structure for counterexample to naive recursive encoding.

LTL formula $\mathbf{F}p$ can be characterized using the following least fixpoint equation: $\mathbf{F}p \equiv p \vee \mathbf{X}\mathbf{F}p$. Therefore, it is tempting to introduce a new boolean variable to encode $\mathbf{F}p$, let us call it a , and encode this fixpoint equation for $k = 0$ as $a_0 \leftrightarrow (p_0 \vee a_0)$. Note that for $k = 0$, the only possible l , the target of the back loop, is $l = 0$ again. Model constraints, actually just the initial state constraint, together with this encoding, result in the following formula:

$$\bar{p}_0 \wedge \lambda_0 \wedge a_0 \wedge (a_0 \leftrightarrow (p_0 \vee a_0))$$

This formula seems to follow the semantics, e.g. the model constraints are enforced and the fixpoint characterization of $\mathbf{F}p$ is encoded. However, it is *satisfiable* by setting a_0 to \top . This should not be the case, since $\mathbf{F}p$ does *not* hold on the single possible path in which p never holds. The problem is that even though this recursive encoding of the LTL formula captures the intention of the fixpoint equation it ignores *least* fixpoint semantics. A similar problem occurs when handling stable models for logic programs [SNS02] or more general in the context of answer set programming (ASP) [Nie99], where the default semantics of recursive properties are defined through least fixpoints. This observation can actually be used positively, in order to succinctly encode bounded witness problem of LTL into ASP [HN03] instead into SAT.

To summarize the example, a naive encoding results in an equation system with cyclic dependencies. A solution to such a system is an arbitrary fixpoint. However, semantics of $\mathbf{F}g$ require a least fixpoint.

The basic idea of the linear encoding in [LBHJ04] is to use several iterations through the fixpoint equations of the LTL subformulas with top most \mathbf{F} operator until the values do not change anymore. As it turns out, two backward iterations are actually enough. For each iteration, the value of an LTL formula at time frame i is encoded with the help of a new boolean variable. This is actually similar to using Tseitin variables [Tse68] to encode propositional formulas of arbitrary structure into CNF.

The variables for the first (inner resp. nested) iteration are written $\langle f \rangle_k^i$, those for the second (outer) iteration $\{f\}_k^i$. The rest of the encoding relates these variables among different subformulas, iterations and time points. The full set of constraints is shown in Fig. 18.3. The correctness of this encoding is established by the following theorem, which is an instance of a more general theorem proved in [BHJ⁺06]:

Theorem 1. *Let f be an LTL formula. If $\{f\}_k^0$ is satisfiable assuming in addition model and looping constraints, then there is a path π with $\pi \models f$.*

$$\begin{aligned}
\{p\}_k^i &\equiv p_i \\
\{\neg p\}_k^i &\equiv \neg p_i \\
\{g \vee h\}_k^i &\equiv \{g\}_k^i \vee \{h\}_k^i \\
\{g \wedge h\}_k^i &\equiv \{g\}_k^i \wedge \{h\}_k^i \\
\langle \mathbf{F}g \rangle_k^i &\equiv \{g\}_k^i \vee \langle \mathbf{F}g \rangle_k^{i+1} & \{\mathbf{F}g\}_k^i &\equiv \{g\}_k^i \vee \langle \mathbf{F}g \rangle_k^{i+1} & \text{if } i < k \\
\langle \mathbf{F}g \rangle_k^i &\equiv \{g\}_k^i & \{\mathbf{F}g\}_k^i &\equiv \{g\}_k^i \vee \bigvee_{l=0}^k (\lambda_l \wedge \langle \mathbf{F}g \rangle_k^l) & \text{if } i = k \\
\langle \mathbf{G}g \rangle_k^i &\equiv \{g\}_k^i \wedge \langle \mathbf{G}g \rangle_k^{i+1} & \{\mathbf{G}g\}_k^i &\equiv \{g\}_k^i \wedge \langle \mathbf{G}g \rangle_k^{i+1} & \text{if } i < k \\
\langle \mathbf{G}g \rangle_k^i &\equiv \{g\}_k^i & \{\mathbf{G}g\}_k^i &\equiv \{g\}_k^i \wedge \bigvee_{l=0}^k (\lambda_l \wedge \langle \mathbf{G}g \rangle_k^l) & \text{if } i = k \\
\langle \mathbf{X}g \rangle_k^i &\equiv \{g\}_k^{i+1} & \{\mathbf{X}g\}_k^i &\equiv \{g\}_k^{i+1} & \text{if } i < k \\
\langle \mathbf{X}g \rangle_k^i &\equiv \bigvee_{l=0}^k (\lambda_l \wedge \{g\}_k^l) & \{\mathbf{X}g\}_k^i &\equiv \bigvee_{l=0}^k (\lambda_l \wedge \{g\}_k^l) & \text{if } i = k
\end{aligned}$$

Figure 18.3. LTL constraints for the linear encoding of LTL into SAT.

For $i < k$ and for each subformula, there are at most k connectives in the outer iteration. For $i = k$, we need $2 \cdot (k + 1)$ binary disjunctions resp. conjunctions. The inner iteration adds k more. Altogether, the encoding is linear in k and the formula size $|f|$.

The previous example shows that one iteration is incorrect, at least for $\mathbf{F}g$. It is interesting to note that encoding the fixpoint equation $\mathbf{G}g \equiv g \wedge \mathbf{X}\mathbf{G}g$ does not suffer from the same problem, due to greatest fixpoint semantics and monotonicity of the encoding. Therefore, it is possible to only use one iteration for \mathbf{G} as it is also clearly the case for the propositional operators and \mathbf{X} , for which we applied this optimization already in Fig. 18.3. More variants, extensions, proofs and experimental results can be found in [BHJ⁺06].

Also note that the “no-lasso case” is also captured by this encoding: The result is satisfiable if there is a finite prefix with $k + 1$ states, such that all infinite paths with this prefix are witnesses.

Another option is to use the “liveness to safety” translation of [BAS02, SB04] which modifies the model, but also will result in a linear encoding of certain LTL formulas and in particular in a linear encoding of fairness constraints.

18.4. Completeness

The encodings of the previous section allow to find witnesses for a particular bound k . If the resulting propositional formula turns out to be satisfiable, we are sure that we have found a witness. If the resulting formula is unsatisfiable we can increase k and search for a longer witness. If the LTL formula has a witness this process will find it. However, if it does not, when should we stop increasing k ?

In this section, we discuss techniques that allow to *terminate* the search with the conclusion that no witness can be found. We focus on the special case of simple safety properties $\mathbf{G}p$, e.g. when searching for a witness of $\mathbf{F}\neg p$.

A first answer was given in the original paper [BCCZ99]. From graph theory we can borrow the concept of diameter, which is the longest shortest path between two nodes resp. states, or more intuitively the maximum distance between connected states. It is also often called eccentricity. If a bad state is reachable, then it is reachable in a shortest path from an initial state, which has length smaller or equal than the diameter.

A number such as the diameter, which allows us to stop BMC and conclude, that no witness can be found, is called completeness threshold (CT) [CKOS04, CKOS05]. Another trivial but in practice almost useless completeness threshold is $|S|$, the number of states.

Furthermore, since a shortest witness for $\mathbf{F}\neg p$ always starts with an initial state and ends in a bad state, where p does not hold, we can also use the following distances as CT: either the largest distance of any reachable state from an initial state or if it is smaller the largest distance of any state to the set of bad states, if it can reach a bad state. The former is also referred to as radius, more specifically as forward radius, the latter as backward radius with respect to the set of bad states.

In practice, the backward radius is often quite small. For instance, if p is inductive ($p_0 \wedge T(s_0, s_1) \Rightarrow p_1$) then the backward radius is 0, because it is impossible to go from a state in which p holds to a state in which p does not hold.

Unfortunately, computing diameters directly is quite hard. It is probably as hard as solving the witness problem in the first place. But there are further weaker CTs, which still are often small enough. The first example is the reoccurrence diameter of [BCCZ99], which is the length of the longest simple path in K . A simple path is another concept borrowed from graph theory and denotes a path, on which all states are different.

The reoccurrence diameter can be arbitrarily larger than the (real) diameter. Consider as example a fully connected graph with n nodes. We can easily generate a simple path of length n without reoccurring state. Since every state is reachable from any other step in one step due to full connectivity, the diameter is 1.

Analogously to the diameter, the forward and backward reoccurrence radii with their obvious definitions are CTs as well. Typically, the forward reoccurrence radius is way too large to be of any practical value, while again there exist many examples where the backward reoccurrence radius is small enough to obtain termination.

The main reason to work with reoccurrence diameters instead of real diameters is the possibility to formulate the former in SAT, while the latter is conjectured to need QBF. A simple path constraint to ensure unique states on a path is as follows:

$$\bigwedge_{0 \leq i < j \leq k} s_i \neq s_j$$

This formulation is quadratic in k . There are multiple solutions to avoid this quadratic overhead. One proposal [KS03, CKOS05] uses hardware implementations of sorting networks, which gives an $O(n \cdot \log^2 n)$ sized encoding, but the complexity of these networks usually results in slower SAT solving times [JB07]. This last paper [JB07] also describes how QBF can be used to encode simple path constraints. Similar results are reported in [DHK05, MVS⁺07].

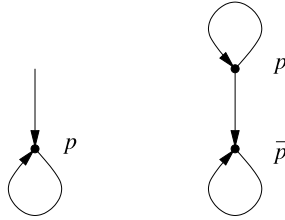


Figure 18.4. Example that k -induction really needs simple path constraints for completeness. The state on the left is the only reachable state, while the two states on the right are not reachable. There exists arbitrary long paths of length k , where p holds on the first k states but not on the last state. However, if these paths are assumed to be simple paths, then the induction step for k -induction becomes unsatisfiable for $k = 2$.

The currently most effective solution is given in [ES03]. The basic idea is to start without any simple path constraints, but then add those inequalities to the SAT solver, which are violated in an assignment returned by the SAT solver. Then the SAT solver is restarted. In practice, the number of incrementally added inequalities is usually very small and even large bounds with many state variables can be handled this way.

18.5. Induction

An important step towards complete BMC techniques was k -induction [SSS00], which also in practice is still quite useful. The basic idea is to strengthen the property p , for which we want to show that $\mathbf{G}p$ holds, by adding more predecessor states. The base case for k -induction is a simple bounded model checking problem:

$$I(s_0) \wedge T(s_0, s_1) \wedge \dots \wedge T(s_{k-1}, s_k) \wedge \neg p_k$$

If the base case is satisfiable, a witness has been found. Otherwise the induction step is checked:

$$p_0 \wedge T(s_0, s_1) \wedge p_1 \wedge T(s_1, s_2) \wedge \dots \wedge p_{k-1} \wedge T(s_{k-1}, s_k) \wedge \neg p_k$$

This is almost a BMC problem, except that the initial state constraint is removed and p is assumed to hold on all states except for the last state. We start with the induction step for $k = 0$, which simply checks whether $\neg p_0$ is unsatisfiable as a propositional formula without any assumptions about the state. If the formula is indeed unsatisfiable, then $\mathbf{G}p$ holds trivially.

Then we check the base case for $k = 0$. If $I(s_0) \wedge \neg p_0$ is satisfiable, then p can be already violated in an initial state. Otherwise we move on to the next induction step at $k = 1$, which is $p_0 \wedge T(s_0, s_1) \wedge \neg p_1$. If this formula is unsatisfiable, we have actually proven that p is inductive for T and again $\mathbf{G}p$ holds.

These two special cases of early termination for BMC, stateless validity and inductiveness, were also discussed in [BCRZ99], but k -induction is able to increase k further and may terminate BMC even with $k > 1$. As the example in

Fig. 18.4 shows this method is not complete. Adding simple path constraints to the induction steps makes it complete, using the argument that the reoccurrence diameter is a CT for plain BMC.

Techniques such as invariant strengthening, discussed further down in Section 18.8, can help to reduce the bound until which k -induction with simple path constraints has to be carried out.

The base case and the induction step share large parts of subformulas even when increasing k to $k + 1$.⁵ To avoid copying these parts and to reuse learned clauses another important technique in this context is the usage of incremental SAT solvers [KWS00, WKS01, ES03]. It is even possible to actively copy learned clauses between time frames as suggested by [Str01]. Variants of k -induction for more general properties are also discussed in [AS06b, HJL05].

18.6. Interpolation

Before Aaron Bradley introduced IC3 [Bra11], model checking based on interpolation [Cra57] was considered to be the most efficient and robust model checking technique, as for instance the comparison in [ADK⁺05] showed. Nowadays IC3 [Bra11] and its variant PDR [EMB11] replaced interpolation based model checking in this regard [CLP⁺14, BvdH17]. Nevertheless interpolation found many applications beside model checking and is still useful in a portfolio approach to model checking. For more information on IC3 see [BK18].

The idea to use interpolation in model checking goes back to Ken McMillan [McM03]. The key idea is to extract an interpolant from a resolution proof for a failed BMC run and use the interpolant as over-approximation for image computation. The extraction algorithm was independently discovered by McMillan, but is very similar to those described in [Kra97, Pud97]. In this section, we present this extraction algorithm and provide a simple straight-forward proof for the propositional case. Our proof is inspired by [McM05] but stripped down to the propositional case. Further usage of interpolation is discussed in Chapter 33 on SMT and in Chapter 20 on Software Verification.

Let A, B be formulas in CNF, c, d clauses, and f, g propositional formulas. With $V(h)$ we denote the set of variables occurring in a formula h . A variable is *global* if it occurs both in A and in B . Let $G = V(A) \cap V(B)$ denote the global set of variables of A and B . A variable is called *local* to A if it only occurs in A and thus is not global. According to [Cra57] a formula f is an *interpolant* for A with respect to B iff it only contains global variables and

$$(I1) \quad A \Rightarrow f \quad \text{and} \quad (I2) \quad B \wedge f \Rightarrow \perp$$

We consider as proof objects *interpolating quadruples* of the form $(A, B) \ c \ [f]$, where the clause c is called the *resolvent* and f the *preliminary interpolant*. Then an interpolating quadruple is *well formed* iff

$$(W1) \quad V(c) \subseteq V(A) \cup V(B) \quad \text{and} \quad (W2) \quad V(f) \subseteq G \cup (V(c) \cap V(A)) \subseteq V(A)$$

⁵Note that p_i can also be assumed in the base case for $i = 0 \dots k - 1$, which would actually be learned by the SAT solver from the previous unsatisfiable base case anyhow.

Thus, an interpolating quadruple is well formed iff the resolvent c is a clause made of literals over variables from A and B , and f only contains variables from A . If in addition, a variable local to A occurs in f , then it also has to occur in c . In approximation to (I1) and (I2) a well formed interpolating quadruple is *valid* iff

$$(V1) \quad A \Rightarrow f \quad \text{and} \quad (V2) \quad B \wedge f \Rightarrow c$$

Note that $A \wedge B \Rightarrow c$ follows immediately from (V1) and (V2). Thus, if a well formed and valid interpolating quadruple with the empty clause \perp as resolvent can be derived, then the preliminary interpolant f of this quadruple actually turns out to be an interpolant of A with respect to B , in particular, $A \wedge B$ is unsatisfiable.

We present tableau rules for deriving well formed and valid interpolating quadruples. This calculus can be interpreted as an annotation mechanism for resolution proofs over $A \wedge B$. It annotates clauses and resolvents with valid interpolating quadruples. Let $c \dot{\vee} l$ denote a clause made up of a subclause c and a literal l , such that $|l|$ does not occur in c , i.e. neither positively nor negatively. The variable of a literal l is written as $|l|$. The proof rules are as follows:

$$(R1) \quad \frac{}{(A, B) c [c]} c \in A \quad \frac{(A, B) c \dot{\vee} l [f] \quad (A, B) d \dot{\vee} \bar{l} [g]}{(A, B) c \vee d [f \wedge g]} |l| \in V(B) \quad (R3)$$

$$(R2) \quad \frac{}{(A, B) c [\top]} c \in B \quad \frac{(A, B) c \dot{\vee} l [f] \quad (A, B) d \dot{\vee} \bar{l} [g]}{(A, B) c \vee d [f|\bar{l} \vee g|\bar{l}]} |l| \notin V(B) \quad (R4)$$

The notation $f|\bar{l}$ denotes the *cofactor* of f with respect to \bar{l} , which is a copy of f , in which occurrences of l are replaced by \perp and occurrences of \bar{l} by \top .

This set of rules simulates the algorithm by McMillan described in [McM03] to extract an interpolant from a resolution refutation, with the exception of rule (R1), the base case for clauses from A . The original algorithm removes variables local to A from the preliminary interpolant immediately. In our first approximation to the algorithm, as given by the tableau rules, we delay the removal until variables local to A are resolved away in (R4), in order to be able to apply an inductive proof argument, i.e. (V2). If one is only interested in the final interpolant of a refutation, where the final resolvent is an empty clause, then it is obviously correct to follow the original algorithm and remove the local variables immediately, since they will be resolved away anyhow. The latter is also the approach taken in [Pud97, YM05], but does not allow to maintain (V1) and (V2).

Theorem 2. *The four rules (R1) – (R4) preserve well formedness and validity.*

Proof. The consequents of the two base case rules (R1) and (R2) are clearly well formed, i.e. (W1) and (W2), and validity, i.e. (V1) and (V2), is easily checked as well. In the inductive case, (W1) also follows immediately. Regarding rules (R3) and (R4), we can assume the antecedents to be well formed and valid.

Let us consider rule (R3) next. The formulas f and g and thus also $f \wedge g$ only contain variables from A . Any variable v of $f \wedge g$ that is local to A is different from $|l|$, since the latter is a variable from B . Therefore, v occurs in c or d and

thus in $c \vee d$, which concludes (W2). The first part (V1) of validity follows from the assumptions, i.e. $A \Rightarrow f$ and $A \Rightarrow g$ obviously imply $A \Rightarrow f \wedge g$. To show the second remaining part (V2) of validity we use soundness of resolution:

$$B \wedge (f \wedge g) \quad \Rightarrow \quad (B \wedge f) \wedge (B \wedge g) \quad \Rightarrow \quad (c \vee l) \wedge (d \vee \bar{l}) \quad \Rightarrow \quad (c \vee d)$$

Proving the consequent of rule (R4) to be well formed, i.e. (W2), is simple: The variable $|l|$ is removed both from the resolvent as well as from the preliminary interpolant of the consequent. In order to show (V1) we can assume $A \Rightarrow f$ and $A \Rightarrow g$. Any assignment σ satisfying A evaluates both formulas $A \rightarrow f$ and $A \rightarrow g$ to \top . Further assume $\sigma(A) = \sigma(l) = \top$. Then Shannon Expansion for g gives $\sigma(g) = \sigma(l \wedge g|_l \vee \bar{l} \wedge g|\bar{l}) = \sigma(g|_l) = \top$. The other case $\sigma(A) = \sigma(\bar{l}) = \top$, with a similar argument, results in $\sigma(f|\bar{l}) = \top$. Thus, $\sigma(f|\bar{l} \vee g|_l) = \top$ for any satisfying assignment σ of A , which concludes part (V1).

The last case (V2) in the proof of the validity of the consequent of rule (R4) is proven as follows. In the assumption $B \wedge f \Rightarrow c \vee l$ we can replace every occurrence of l by \perp respectively every occurrence of \bar{l} by \top without making the assumption invalid. This implies $B \wedge f|\bar{l} \Rightarrow c$, since $|l|$ does not occur in B , nor in c . With a similar argument we obtain $B \wedge g|_l \Rightarrow d$ and thus $B \wedge (f|\bar{l} \vee g|_l) \Rightarrow (c \vee d)$. This completes the proof of the validity of the consequent of rule (R4) assuming validity of its antecedents and concludes the whole proof. \square

The extended version of [YM05], which was published as a technical report [YM04] proves a variant of the algorithm given in [Pud97]. Their inductive argument is slightly more complicated than ours, i.e. (V1) and (V2). In addition, our formulation allows to derive an relation between A , B , c and its preliminary interpolant f . In particular, our preliminary interpolant f captures enough information from A , such that B together with f implies c .

The strongest interpolant of A is obtained from A by existentially quantifying over all local variables in A . Thus, interpolation can be seen as an over approximation of quantifier elimination. Consider the following example, where A contains the four clauses A_0 to A_3

$$A_0 : a \vee c \vee d \quad A_1 : \bar{a} \vee c \vee d \quad A_2 : a \vee \bar{c} \vee \bar{d} \quad A_3 : \bar{a} \vee \bar{c} \vee \bar{d}$$

and B the following four clauses B_0 to B_3 :

$$B_0 : b \vee \bar{c} \quad B_1 : \bar{b} \vee \bar{c} \quad B_2 : b \vee \bar{d} \quad B_3 : \bar{b} \vee \bar{d}$$

Variable a is local to A , while b is local to B , and the other variables c and d are global. Quantifying a from A results in $\exists a[A] \equiv c \oplus d$, where “ \oplus ” is the XOR operator, i.e. c and d have different value. Since quantifying b from B results in $\exists b[B] \equiv \bar{c} \wedge \bar{d}$, i.e. c and d both have to be \perp and are thus forced to the same

value, the CNF $A \wedge B$ is unsatisfiable. A possible refutation proof is as follows:

$Q_0 :$	(A, B)	$c \vee d$	$[c \vee d]$	resolved from A_0 and A_1
$Q_1 :$	(A, B)	\bar{c}	$[\top]$	resolved from B_0 and B_1
$Q_2 :$	(A, B)	d	$[c \vee d]$	resolved from Q_0 and Q_1
$Q_3 :$	(A, B)	\bar{d}	$[\top]$	resolved from B_2 and B_3
$Q_4 :$	(A, B)	\perp	$[c \vee d]$	resolved from Q_2 and Q_3

In brackets, we list the partial interpolants. The final interpolant of A with respect to B is $P \equiv c \vee d$, which is weaker than the strongest interpolant $\exists a[A]$, i.e. $\exists a[A] \Rightarrow P$, but it exactly captures the part of A which is required for the refutation: either c or d has to be \top .

18.7. Completeness with Interpolation

It has been shown in [McM03] that termination checks for bounded model checking do not need exact quantifier elimination algorithms as has been previously assumed [WBCG00, ABE00, AB02, MS03, McM02, PBZ03]. An over approximation as given by an interpolant extracted from a refutation of a failed bounded model checking run is enough.

The resulting method of [McM03], which we describe in this section, is originally restricted to simple safety properties. But using the “liveness to safety” translation of [BAS02, SB04] it can be applied to more general specifications, which also is reported to work reasonably well in practice.

Assume that we want to prove that p is not reachable, i.e. the property $\mathbf{G}p$ holds, or respectively there is no witness for $\mathbf{F}\neg p$. Let k be the backward radius with respect to $\neg p$. More precisely let k be the smallest number such that all states which can reach $\neg p$, can reach a state in which $\neg p$ holds in k' steps, where $k' \leq k$. The backward radius can be interpreted as the number of generations in backward breadth first search (BFS), starting with the set of bad states in which $\neg p$ holds until all states that can reach a bad state are found. In BDD-based model checking [McM93] this is the number of *preimage* computations until a fixpoint is reached, starting with the set of bad states.

If the property holds and no witness to its negation exists, then the following formula is unsatisfiable (after checking $I(s_0) \Rightarrow p_0$ separately):

$$\underbrace{I(s_0) \wedge T(s_0, s_1)}_A \wedge \underbrace{T(s_1, s_2) \wedge \dots \wedge T(s_k, s_{k+1}) \wedge \bigvee_{j=1}^{k+1} \neg p_j}_B$$

Note that we “unroll” one step further as usual. Let P_1 be the interpolant of A with respect to B . Since $P_1 \wedge B$ is unsatisfiable, all states that satisfy P_1 cannot reach a bad state in k or fewer steps. As a generalization consider the following

sequence of formulas:

$$F_i \quad : \quad \underbrace{R_i(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge \dots \wedge T(s_k, s_{k+1})}_{A_i} \wedge \underbrace{\bigvee_{j=1}^{k+1} \neg p_j}_{B_i}$$

These are all BMC problems with the same bound $k+1$. In the base case, let $P_0 = R_0 = I$. In order to define R_i , under the assumption that F_i is unsatisfiable, let P_{i+1} be the interpolant of A_i with respect to B_i and $R_{i+1}(s) = R_i(s) \vee P_{i+1}[s_0/s_1]$, where $P_{i+1}[s_0/s_1]$ is obtained from P_{i+1} by replacing s_1 with s_0 . If F_i becomes satisfiable, then P_j , R_j and F_j are undefined for all $j > i$.

With the same argument as in the base case, we can show that as long P_i is defined all states satisfying R_i cannot reach a bad state in k or less steps, or to phrase it differently: R_i -states are more than k steps away from bad states.

Now let us assume that there exists a smallest i for which F_i is satisfiable. Then there is a state s_0 which reaches a bad state in $k+1$ or less steps but does not reach a bad state in k or less steps. The former just immediately follows from F_i being satisfied, the latter from s_0 satisfying R_i and thus s_0 being at least $k+1$ steps away from bad states. However, this contradicts our assumption that k is the backward radius and there are no states that need more than k steps to reach a bad state. Thus, P_i , R_i and F_i are defined for all $i \in \mathbb{N}$.

In addition, since $R_i \Rightarrow R_{i+1}$, we have an increasing chain of weaker and weaker starting points, which for a finite model has to reach a fixpoint. Therefore, there is an n for which $R_n \equiv R_{n+j}$ for all $j \in \mathbb{N}$. As soon $P_{n+1} \Rightarrow R_n$, which can be checked by a SAT solver, R_n is inductive, is implied by $I(s_0)$ and is stronger than p . Therefore, we can stop and conclude $\mathbf{G}p$ to hold resp. that no witness to $\mathbf{F}\neg p$ exists. This conclusion is correct even if k is smaller than the backward radius and all F_i are defined, i.e. if there exists an n for which $P_{n+1} \Rightarrow R_n$.

However, if k is smaller than the backward radius it may happen that F_i is satisfiable for $i > 0$. In this case, we simply increase k and start a new sequence of F_j 's. In any case, if F_0 ever turns out to be satisfiable, then we have found a witness, which is a counterexample to $\mathbf{G}p$.

To implement this algorithm, a SAT solver is required which is able to generate resolution proofs. This feature is easy to implement on top of DPLL style solvers, particularly for those that use learning. The overhead to produce resolution proofs is in general acceptable [ZM03, Gel07]. However, interpolants tend to be highly redundant [McM03]. In practice, it is necessary to shrink their size with circuit optimization techniques such as SAT sweeping [Kue04] and AIG rewriting [MCB06].

The outer loop increases k until either the BMC problem becomes satisfiable or the inner loop terminates because $P_{n+1} \Rightarrow R_n$ holds. As we have shown, k is bounded by the backward radius and thus it is beneficial to strengthen p as much as possible to decrease the backward radius which in order reduces not only the number of iterations of the outer loop but also the size (at least the length) of the BMC problems.

18.8. Invariant Strengthening

If a property p is inductive ($p_0 \wedge T(s_0, s_1) \Rightarrow p_1$), then a BMC run for $k = 1$ without initial state constraints is unsatisfiable, and proves that p holds in all states, i.e. $K \models \mathbf{G}p$, unless p is violated in the initial state.

In general, it is difficult to come up with strong enough inductive invariants. However, even if p does not imply the real invariant q , which we want to prove, p 's inductiveness can still be used to strengthen q . Then we can try to prove $\mathbf{G}(p \wedge q)$ instead of $\mathbf{G}q$. The former often has a smaller backward radius, in particular the backward radius never increases after strengthening, and can help to terminate k -induction and interpolation earlier. This is particularly useful for k -induction, which suffers from an exponential gap between backward recurrence radius and real backward radius.

In the context of sequential equivalence checking, useful invariants are of course equalities between signals. If such an equality $\mathbf{G}(p = q)$ between two signals p and q is suspected, then we can try to check whether $p = q$ is inductive. This idea can be extended to multiple signal pairs, e.g. $\mathbf{G} \bigwedge_{j=1}^n (p^j = q^j)$. In this case, inductiveness is proven if the following SAT problems for $m = 1 \dots n$ are all unsatisfiable:

$$\bigwedge_{j=1}^n (p_0^j = q_0^j) \wedge T(s_0, s_1) \wedge (p_1^m \neq q_1^m)$$

This idea is described in [vE98] and has been extended to property checking [BC00, CNQ07, AS06a, BM07] and can also make use of k -induction (see Section 18.5). Related to adding invariants is target enlargement [BKA02, BBC⁺05], which increases the set of bad resp. target states by some states that provably can reach a bad state.

18.9. Related Work

In the late 90ties, the performance of SAT solvers increased considerably [MSS99, Bor97, Zha97]. At the same time progress in BDDs stalled. It became clear that BDD-based symbolic model checking cannot handle more than a couple of hundred latches, which is much smaller than what most industrial applications require. This was the main motivation behind trying to apply SAT technology to model checking. The first angle of attack was to use QBF solvers, because these allow to solve the same problem as BDD-based model checking. However, at that time, QBF solvers were lagging behind SAT solvers. The first real implementations just started to emerge [CGS98]. Therefore, a paradigm shift was necessary.

The development of BMC was influenced by SAT-based planning [KS92]. See also Chapter 19 in this handbook, which is devoted to SAT-based planning. For simple safety properties, the tasks are similar: try to find a bounded witness, e.g. a *plan*, which reaches the goal resp. the bad state. The main contribution of BMC was to show how this idea can be lifted to infinite paths and thus produce witnesses for arbitrary temporal formulas. Furthermore initial attempts were

made to prove properties. Of course, the major difficulty was to convince the formal verification community that a focus on falsification can be beneficial.

Deciding QBF plays the same role for PSPACE-hard problems as SAT does for NP hard problems. Since symbolic model checking is PSPACE complete as well [Sav70], see [PBG05] for more details, it seems natural to use QBF solvers for symbol model checking, as it was already proposed in the original BMC paper [BCCZ99]. However, even though QBF solving is improving, there are very few successful applications of QBF to symbolic model checking [DHK05, CKS07, MVS⁺07]. Most results are negative [JB07].

Often properties are local and can be proven locally. A standard technique in this context is automatic abstraction refinement. Initially, the model checker abstracts the system by removing all the model constraints on variables apart from those that directly occur in the property. This abstraction is conservative in the following sense: if the property holds in the abstracted model, then it also holds in the concrete model. If it does not hold and the abstract counterexample cannot be mapped to the concrete model the abstraction has to be refined by adding back variables and model constraints. There are various techniques that use BMC in this context. They either use proofs and unsatisfiable cores [MA03] or follow the counterexample-guided abstraction refinement (CEGAR) paradigm [CGJ⁺03]. For further details, see [PBG05] and particularly Chapter 20. We also skipped most material on circuit based techniques, including quantifier elimination [WBCG00, ABE00, AB02, McM02, PBZ03, PSD06] circuit cofactoring [GGA04] and ATPG-based techniques, which are also discussed in [PBG05].

Finally, BMC has been extended to more general models, including software as discussed in Chapter 20. BMC is used for infinite systems [dMRS03], more specifically for hybrid systems [ABCS05, FH05]. In this context, bounded semantics are typically decidable, while the general model checking problem is not. Nevertheless, complete techniques, such as k -induction and interpolation can still be useful and allow to occasionally prove properties.

18.10. Conclusion

The main reason behind the success of BMC, is the tremendous increase in reasoning power of recent SAT solvers, particularly the breakthrough realized by Chaff [MMZ⁺01] right two years after the first publication on BMC. SAT and BMC became a standard tool in the EDA industry thereafter. Their importance will be emphasized as SAT solver's capacity continues to increase.

Acknowledgements

The author would like to thank Keijo Heljanko, Viktor Schuppan and Daniel Kröning for their very valuable comments on drafts of this chapter for the first edition of the handbook.

References

- [AB02] A. Ayari and D. A. Basin. QUBOS: Deciding quantified boolean logic using propositional satisfiability solvers. In M. Aagaard and J. W. O’Leary, editors, *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design (FMCAD’02)*, volume 2517 of *Lecture Notes in Computer Science*, pages 187–201. Springer, 2002.
- [ABCS05] G. Audemard, M. Bozzano, A. Cimatti, and R. Sebastiani. Verifying industrial hybrid systems with MathSAT. *Electronic Notes in Theoretical Computer Science*, 119(2):17–32, 2005. In *Proceedings of the 2nd International Workshop on Bounded Model Checking (BMC’04)*.
- [ABE00] P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic Reachability Analysis Based on SAT-Solvers. In S. Graf and M. Schwartzbach, editors, *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’00)*, volume 1785 of *Lecture Notes in Computer Science*, pages 411–425. Springer, 2000.
- [ADK⁺05] N. Amla, X. Du, A. Kuehlmann, R. P. Kurshan, and K. L. McMillan. An analysis of SAT-based model checking techniques in an industrial environment. In D. Borrione and W. J. Paul, editors, *Proceedings of 13th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME’05)*, volume 3725 of *Lecture Notes in Computer Science*, pages 254–268, 2005.
- [AS06a] M. Awedh and F. Somenzi. Automatic invariant strengthening to prove properties in bounded model checking. In *Proceedings of the 43rd Design Automation Conference (DAC’06)*, pages 1073–1076. ACM, 2006.
- [AS06b] M. Awedh and F. Somenzi. Termination criteria for bounded model checking: Extensions and comparison. *Electronic Notes in Theoretical Computer Science*, 144(1):51–66, 2006. In *Proceedings of the 3rd International Workshop on Bounded Model Checking (BMC’05)*.
- [BAS02] A. Biere, C. Artho, and V. Schuppan. Liveness Checking as Safety Checking. *Electronic Notes in Theoretical Computer Science*, 66(2), 2002. In *Proceedings of the 7th International Workshop on Formal Methods for Industrial Critical Systems (FMICS’02)*.
- [BBC⁺05] G. P. Bischoff, K. S. Brace, G. Cabodi, S. Nocco, and S. Quer. Exploiting target enlargement and dynamic abstraction within mixed BDD and SAT invariant checking. *Electronic Notes in Theoretical Computer Science*, 119(2):33–49, 2005. In *Proceedings of the 2nd International Workshop on Bounded Model Checking (BMC’04)*.
- [BC00] P. Bjesse and K. Claessen. SAT-based Verification without State Space Traversal. In W. A. H. Jr. and S. D. Johnson, editors, *Proceedings of the 3rd International Conference on Formal Methods in Computer Aided Design (FMCAD’00)*, volume 1954 of *Lecture Notes in Computer Science*, pages 372–389. Springer, 2000.

- [BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In R. Cleaveland, editor, *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.
- [BCM⁺92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.
- [BCRZ99] A. Biere, E. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs. In N. Halbwachs and D. Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 1999.
- [BHJ⁺06] A. Biere, K. Heljanko, T. A. Junttila, T. Latvala, and V. Schuppan. Linear encodings of bounded LTL model checking. *Logical Methods in Computer Science (LMCS'06)*, 2(5:5), 2006.
- [BK18] A. Biere and D. Kroening. Sat-based model checking. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking.*, pages 277–303. Springer, 2018.
- [BKA02] J. Baumgartner, A. Kuehlmann, and J. A. Abraham. Property Checking via Structural Analysis. In E. Brinksma and K. G. Larsen, editors, *Proceedings of the 14th International Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2002.
- [BM07] A. R. Bradley and Z. Manna. Checking safety by inductive generalization of counterexamples to induction. In *Proceedings of 7th International Conference on Formal Methods in Computer-Aided Design (FMCAD'07)*, pages 173–180. IEEE Computer Society, 2007.
- [Bor97] A. Borålv. The industrial success of verification tools based on Stålmarck's method. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 7–10. Springer, 1997.
- [Bra11] A. R. Bradley. Sat-based model checking without unrolling. In R. Jhala and D. A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011.
- [Bry86] R. E. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C(35), 1986.
- [BvDH17] A. Biere, T. van Dijk, and K. Heljanko. Hardware model checking competition 2017. In D. Stewart and G. Weissenbacher, editors, *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, page 9. IEEE, 2017.
- [CE82] E. M. Clarke and A. Emerson. Design and Synthesis of Synchroniza-

- tion Skeletons Using Branching-Time Temporal Logic. In *Proceedings of the Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*. Springer, 1982.
- [CGH97] E. M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design.*, 10(1):47–71, 1997.
- [CGJ+03] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5), 2003.
- [CGP99] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT press, 1999.
- [CGS98] M. Cadoli, A. Giovanardi, and M. Schaerf. An Algorithm to Evaluate Quantified Boolean Formulae. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98)*, pages 262–267, 1998.
- [CKOS04] E. M. Clarke, D. Kroening, J. Ouaknine, and O. Strichman. Completeness and complexity of bounded model checking. In B. Steffen and G. Levi, editors, *Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'04)*, volume 2937 of *Lecture Notes in Computer Science*, pages 85–96. Springer, 2004.
- [CKOS05] E. M. Clarke, D. Kroening, J. Ouaknine, and O. Strichman. Computational challenges in bounded model checking. *Software Tools for Technology Transfer (STTT)*, 7(2):174–183, 2005.
- [CKS07] B. Cook, D. Kroening, and N. Sharygina. Verification of boolean programs with unbounded thread creation. *Theoretical Computer Science*, 388(1-3):227–242, 2007.
- [CLP+14] G. Cabodi, C. Loiacono, M. Palena, P. Pasini, D. Patti, S. Quer, D. Vendraminetto, A. Biere, and K. Heljanko. Hardware model checking competition 2014: An analysis and comparison of solvers and benchmarks. *JSAT*, 9:135–172, 2014.
- [CNQ07] G. Cabodi, S. Nocco, and S. Quer. Boosting the role of inductive invariants in model checking. In R. Lauwereins and J. Madsen, editors, *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'07)*, pages 1319–1324. ACM, 2007.
- [Cra57] W. Craig. Linear reasoning: A new form of the Herbrand-Gentzen theorem. *Journal of Symbolic Logic*, 22(3):250–268, 1957.
- [DHK05] N. Dershowitz, Z. Hanna, and J. Katz. Bounded model checking with QBF. In F. Bacchus and T. Walsh, editors, *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, volume 3569 of *Lecture Notes in Computer Science*. Springer, 2005.
- [dMRS03] L. M. de Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: From refutation to verification. In W. A. H. Jr. and F. Somenzi, editors, *Proceedings of the 15th International Conferences on Computer Aided Verification (CAV'03)*, volume 2725 of *Lecture Notes in Computer Science*, pages 14–26. Springer, 2003.
- [EMB11] N. Eén, A. Mishchenko, and R. K. Brayton. Efficient implementation

- of property directed reachability. In P. Bjesse and A. Slobodová, editors, *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, pages 125–134. FMCAD Inc., 2011.
- [Eme90] E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. MIT Press, 1990.
- [ES03] N. Eén and N. Sörensson. Temporal Induction by Incremental SAT Solving. *Electronic Notes in Theoretical Computer Science*, 89(4), 2003. In *Proceedings of the 1st International Workshop on Bounded Model Checking (BMC'03)*.
- [FH05] M. Fränzle and C. Herde. Efficient proof engines for bounded model checking of hybrid systems. *Electronic Notes in Theoretical Computer Science*, 133:119–137, 2005.
- [FSW02] A. M. Frisch, D. Sheridan, and T. Walsh. A fixpoint encoding for bounded model checking. In *Proceedings of the 4th International Conference on Formal Methods in Computer Aided Design (FMCAD'02)*, volume 2517 of *Lecture Notes in Computer Science*, pages 238–255. Springer, 2002.
- [Gel07] A. V. Gelder. Verifying propositional unsatisfiability: Pitfalls to avoid. In J. Marques-Silva and K. A. Sakallah, editors, *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing (SAT'07)*, volume 4501 of *Lecture Notes in Computer Science*, pages 328–333. Springer, 2007.
- [GGA04] M. K. Ganai, A. Gupta, and P. Ashar. Efficient SAT-based unbounded symbolic model checking using circuit cofactoring. In *Proceedings International Conference on Computer-Aided Design (ICCAD'04)*, pages 510–517. IEEE Computer Society / ACM, 2004.
- [Hel01] K. Heljanko. Bounded reachability checking with process semantics. In K. G. Larsen and M. Nielsen, editors, *Proceedings of the 12th International Conference on Concurrency Theory (CONCUR'01)*, volume 2154 of *Lecture Notes in Computer Science*, pages 218–232. Springer, 2001.
- [HJK⁺06] K. Heljanko, T. A. Junttila, M. Keinänen, M. Lange, and T. Latvala. Bounded model checking for weak alternating Büchi automata. In T. Ball and R. B. Jones, editors, *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 95–108. Springer, 2006.
- [HJL05] K. Heljanko, T. A. Junttila, and T. Latvala. Incremental and complete bounded model checking for full PLTL. In K. Etessami and S. K. Rajamani, editors, *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 98–111. Springer, 2005.
- [HN03] K. Heljanko and I. Niemelä. Bounded LTL model checking with stable models. *Theory and Practice of Logic Programming*, 3(4-5):519–550, 2003.
- [Hol04] G. Holzmann. *The SPIN Model Checker*. Addison Wesley, 2004.
- [JB07] T. Jussila and A. Biere. Compressing BMC encodings with QBF.

- Electronic Notes in Theoretical Computer Science*, 174(3):45–56, 2007. In *Proceedings of the 4th International Workshop on Bounded Model Checking (BMC'06)*.
- [JHN03] T. Jussila, K. Heljanko, and I. Niemelä. BMC via on-the-fly determinization. *Electronic Notes in Theoretical Computer Science*, 89(4), 2003. In *Proceedings of the 1st International Workshop on Bounded Model Checking (BMC'03)*.
- [KK97] A. Kuehlmann and F. Krohm. Equivalence checking using cuts and heaps. In *Proceedings of the 34th Design Automation Conference (DAC'97)*, pages 263–268. ACM, 1997.
- [KPKG02] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai. Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(12):1377–1394, 2002.
- [Kra97] J. Krajíček. Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. *Journal on Symbolic Logic*, 62(2):457–486, 1997.
- [KS92] H. Kautz and B. Selman. Planning as satisfiability. In B. Neumann, editor, *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363. John Wiley & Sons, 1992.
- [KS03] D. Kroening and O. Strichman. Efficient computation of recurrence diameters. In L. D. Zuck, P. C. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'03)*, volume 2575 of *Lecture Notes in Computer Science*, pages 298–309. Springer, 2003.
- [Kue04] A. Kuehlmann. Dynamic transition relation simplification for bounded property checking. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'04)*, pages 50–57. IEEE Computer Society / ACM, 2004.
- [Kur08] R. P. Kurshan. Verification technology transfer. In H. Veith and O. Grumberg, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 46–64. Springer, 2008.
- [KWS00] J. Kim, J. Whitemore, and K. Sakallah. On Solving Stack-Based Incremental Satisfiability Problems. In *Proceedings of the International Conference on Computer Design (ICCD'00)*, pages 379–382, 2000.
- [LBHJ04] T. Latvala, A. Biere, K. Heljanko, and T. A. Junttila. Simple bounded LTL model checking. In *Proceedings of the 6th International Conference on Formal Methods in Computer Aided Design (FMCAD'04)*, volume 3312 of *Lecture Notes in Computer Science*, pages 186–200. Springer, 2004.
- [LP85] O. Lichtenstein and A. Pnueli. Checking that finite state programs satisfy their linear specification. In *ACM Symposium on Principles of Programming Languages*, pages 97–107, 1985.
- [MA03] K. L. McMillan and N. Amla. Automatic abstraction without counterexamples. In H. Garavel and J. Hatcliff, editors, *Proceedings of the International Conference on Tools and Algorithms for the Construc-*

- tion and Analysis of Systems (TACAS'03)*, volume 2619 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2003.
- [Mai00] M. Mairdl. *Using model checking for system verification*. PhD thesis, Ludwig-Maximilians-Universität at München, 2000.
- [MCB06] A. Mishchenko, S. Chatterjee, and R. K. Brayton. DAG-aware AIG rewriting a fresh look at combinational logic synthesis. In E. Sentovich, editor, *Proceedings of the 43rd Design Automation Conference (DAC'06)*, pages 532–535. ACM, 2006.
- [McM93] K. L. McMillan. *Symbolic Model Checking: An approach to the State Explosion Problem*. Kluwer, 1993.
- [McM02] K. L. McMillan. Applying SAT Methods in Unbounded Symbolic Model Checking. In E. Brinksma and K. G. Larsen, editors, *Proceedings of the 14th International Conference on Computer-Aided Verification (CAV'02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 250–264. Springer, 2002.
- [McM03] K. L. McMillan. Interpolation and SAT-based Model Checking. In J. W. A. Hunt and F. Somenzi, editors, *Proceedings of the 15th Conference on Computer-Aided Verification (CAV'03)*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.
- [McM05] K. L. McMillan. An interpolating theorem prover. *Theoretical Computer Science*, 345(1), 2005.
- [MMZ⁺01] M. H. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535, 2001.
- [MS03] M. Mneimneh and K. Sakallah. SAT-based sequential depth computation. In *Proceedings of the Asia South Pacific Design Automation Conference (ASPDAC'03)*, pages 87–92. ACM, 2003.
- [MSS99] J. P. Marques-Silva and K. A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.
- [MVS⁺07] H. Mangassarian, A. G. Veneris, S. Safarpour, M. Benedetti, and D. Smith. A performance-driven QBF-based iterative logic array representation with applications to verification, debug and test. In G. G. E. Gielen, editor, *Proceedings of the International Conference on Computer-Aided Design (ICCAD'07)*, pages 240–245. IEEE, 2007.
- [Nie99] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.
- [PBG05] M. Prasad, A. Biere, and A. Gupta. A survey on recent advances in SAT-based formal verification. *Software Tools for Technology Transfer (STTT)*, 7(2), 2005.
- [PBZ03] D. Plaisted, A. Biere, and Y. Zhu. A Satisfiability Procedure for Quantified Boolean Formulae. *Discrete Applied Mathematics*, 130(2):291–328, 2003.
- [Pel94] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Proceedings of the 6th International Conference on Computer Aided Verification (CAV'94)*, pages 377–390. Springer-Verlag,

- 1994.
- [Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of IEEE Symposium on Foundations of Computer Science*, 1977.
 - [PSD06] F. Pigorsch, C. Scholl, and S. Disch. Advanced unbounded model checking based on AIGs, BDD sweeping, and quantifier scheduling. In *Proceedings 6th International Conference on Formal Methods in Computer-Aided Design (FMCAD'06)*, pages 89–96. IEEE Computer Society, 2006.
 - [Pud97] P. Pudlák. Lower bounds for resolution and cutting planes proofs and monotone computations. *Journal of Symbolic Logic*, 62(3), 1997.
 - [QS82] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*. Springer, 1982.
 - [Sav70] W. J. Savitch. Relational between nondeterministic and deterministic tape complexity. *Journal of Computer and System Sciences*, 4:177–192, 1970.
 - [SB04] V. Schuppan and A. Biere. Efficient reduction of finite state model checking to reachability analysis. *Software Tools for Technology Transfer (STTT)*, 5(1-2):185–204, 2004.
 - [SC85] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.
 - [SNS02] P. Simons, I. Niemelä, and T. Soinen. Extending and implementing the stable model semantics. *Journal on Artificial Intelligence*, 138(1-2):181–234, 2002.
 - [SSS00] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In W. A. H. Jr. and S. D. Johnson, editors, *Proceedings of the 3rd International Conference on Formal Methods in Computer Aided Design (FMCAD'00)*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2000.
 - [Str01] O. Strichman. Pruning Techniques for the SAT-based Bounded Model Checking Problem. In T. Margaria and T. F. Melham, editors, *Proceedings of the 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'01)*, volume 2144 of *Lecture Notes in Computer Science*, pages 58–70. Springer, 2001.
 - [Tse68] G. S. Tseitin. On the Complexity of Derivation in Propositional Calculus. In *Studies in Constructive Mathematics and Mathematical Logic, Part II*, volume 8 of *Seminars in Mathematics*, pages 234–259. V.A. Steklov Mathematical Institute, 1968. English Translation: Consultants Bureau, New York, 1970, pages 115 – 125.
 - [Var08] M. Y. Vardi. From Church and Prior to PSL. In H. Veith and O. Grumberg, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 150–171. Springer, 2008.
 - [vE98] C. A. J. van Eijk. Sequential equivalence checking without state space traversal. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'98)*, pages 618–623. IEEE Computer So-

- ciety, 1998.
- [VG08] H. Veith and O. Grumberg, editors. *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*. Springer, 2008.
 - [VW94] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.
 - [WBCG00] P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining Decision Diagrams and SAT Procedures for Efficient Symbolic Model Checking. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 2000.
 - [WKS01] J. P. Whittimore, J. Kim, and K. A. Sakallah. SATIRE: A New Incremental Satisfiability Engine. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 542–545, 2001.
 - [WVS83] P. Wolper, M. Vardi, and A. P. Sistla. Reasoning about infinite computation paths. In *Proceedings of 24th Annual Symposium on Foundations of Computer Science (FOCS'83)*, pages 185–194. IEEE Computer Society, 1983.
 - [YM04] G. Yorsh and M. Musuvathi. A combination method for generating interpolants. Technical Report MSR-TR-2004-108, Microsoft Research, 2004.
 - [YM05] G. Yorsh and M. Musuvathi. A combination method for generating interpolants. In R. Nieuwenhuis, editor, *Proceedings of the 20th International Conference on Automated Deduction (CADE'05)*, volume 3632 of *Lecture Notes in Computer Science*, pages 353–368. Springer, 2005.
 - [Zha97] H. Zhang. SATO: An efficient propositional prover. In W. McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction (CADE'97)*, volume 1249 of *Lecture Notes in Computer Science*, pages 272–275. Springer, 1997.
 - [ZM03] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'03)*, pages 10880–10885. IEEE Computer Society, 2003.

Chapter 19

Planning and SAT

Jussi Rintanen

Planning in Artificial Intelligence is one of the earliest applications of SAT to solving generic search problems. The planning problem involves finding a sequence of actions that reaches a given goal. Such an action sequence and an associated state sequence correspond to a satisfying assignment of a propositional formula which can be easily constructed from the problem description.

19.1. Introduction

Planning in artificial intelligence is the process of finding a sequence of actions that reaches a predefined goal. The problem is usually expressed in terms of a description of an initial state, a set of actions, and a goal state. In the most basic form of planning, called classical planning, there is only one initial state and the actions are deterministic.

In their 1992 paper, Kautz and Selman suggested that the classical planning problem could be solved by translating it into a propositional formula and testing its satisfiability [KS92]. This approach was in strong contrast with earlier works on planning which viewed planning as a deductive problem or a specialized search problem. At first, this idea did not attract much interest, but in 1996 Kautz and Selman demonstrated that the best algorithms for SAT together with improved translations provided a competitive approach to classical planning [KS96].

The success of SAT in planning suggested the same approach to solving other similar problems, such as model-checking in computer-aided verification and validation. A SAT-based approach to model-checking properties expressed in the linear temporal logic LTL [Pnu77] was proposed soon after the success of planning as satisfiability became more widely known [BCCZ99].

Following the work by Kautz and Selman, similar translations of planning into many other formalisms were proposed: nonmonotonic logic programs [DNK97], mixed integer linear programming [KW99, WW99, VBLN99], and constraint satisfaction problems CSP [vBC99, DK01].

The classical planning problem is PSPACE-complete [Byl94], so there are presumably (assuming $P \neq PSPACE$) no polynomial time translations from classical planning into SAT. The formulae that represent the classical planning problem

may, in the worst case, be exponential in the size of the problem instance. However, the formula size is determined by a natural parameter which tends not to be very high in practically interesting cases: the plan length. Most classes of planning problems are solvable with plans of a polynomial length, which means that the corresponding propositional formulae have a polynomial size. So in practice it is feasible to reduce the planning problem to SAT.

The underlying idea in using SAT for planning is to encode the bounded plan existence problem, i.e. whether a plan of a given length n exists, as a formula in the classical propositional logic. The formula for a given n is satisfiable if and only if there is a plan of size n . Finding a plan reduces to testing the satisfiability of the formulae for different values of n .

The efficiency of the SAT-based approach is determined by three largely orthogonal components.

1. Efficient representations ϕ_n of the planning problem for a given plan length n . This is the topic of Sections 19.3 and 19.4.
2. Efficient algorithms for testing the satisfiability of ϕ_n .
3. Algorithms for choosing which values of n to consider to find a satisfiable ϕ_n as quickly as possible. This is the topic of Section 19.5.

Section 19.3 describes the most basic representation of planning in SAT, and the most important improvements to it to achieve efficient planning.

Section 19.4 introduces the notion of parallel plans [BF97, KS96] which is an important factor in the efficiency of planning as satisfiability. In a parallel plan there may be more than one action at each time point. The length parameter n restricts the number of time points but not directly the number of actions. Different notions of parallel plans can be defined, and for maintaining the connection to sequential plans it is required that there is a parallel plan exactly when there is a sequential plan, and moreover, mapping a given parallel plan to a sequential plan should be possible in polynomial time.

Section 19.5 presents different strategies for making the satisfiability tests for the bounded planning problem for different lengths.

Section 19.6 describes the extension of Kautz and Selman's ideas to temporal planning, in which actions have a real or rational valued duration and multiple actions can temporally each other.

Section 19.7 discusses possibilities to reduce more general planning problems, for example with nondeterminism and sensing, to SAT and its quantified extensions.

19.2. Classical Planning

We consider planning in a setting where the states of the world are represented in terms of a set X of Boolean state variables which take the value *true* or *false*. Formulae are formed from the state variables with the connectives \vee and \neg . The connectives \wedge , \rightarrow and \leftrightarrow are defined in terms of \vee and \neg . A *literal* is a formula of the form x or $\neg x$ where $x \in X$ is a state variable. We define the *complements* of literals as $\bar{x} = \neg x$ and $\overline{\bar{x}} = x$ for all $x \in X$. A *clause* is a disjunction $l_1 \vee \dots \vee l_n$ of one or more literals. We also use the constant atoms \top and \perp for denoting

true and false, respectively. Each state $s : X \rightarrow \{0, 1\}$ assigns each state variable in X a value 0 or 1.

Actions change the state of the world.

Definition 19.2.1. An *action* over a set of state variables X is a pair $\langle p, e \rangle$ where

1. p is a propositional formula over X (*the precondition*) and
2. e is a set of pairs $f \Rightarrow d$ (*the effects*) where f is a propositional formula over X and d is a set of literals over X . (In the planning literature, the case in which f is not \top is called a “conditional effect”.)

The meaning of $f \Rightarrow d$ is that the literals in d are made true if the formula f is true. For an action $a = \langle p, e \rangle$ its *active effects* in a state s are

$$[a]_s = [e]_s = \bigcup \{d \mid f \Rightarrow d \in e, s \models f\}.$$

The action is *executable* in s if $s \models p$ and $[a]_s$ is consistent (does not contain both x and $\neg x$ for any $x \in X$.) If this is the case, then we define $exec_a(s)$ as the unique state that is obtained from s by making $[a]_s$ true and retaining the values of the state variables not occurring in $[a]_s$. For sequences $a_1; a_2; \dots; a_n$ of actions we define $exec_{a_1; a_2; \dots; a_n}(s)$ as $exec_{a_n}(\dots exec_{a_2}(exec_{a_1}(s)) \dots)$. For sets S of actions and states s we define $exec_S(s)$ as the result of simultaneously executing all actions in S . We require that $exec_a(s)$ is defined for every $a \in S$ and that the set $[S]_s = \bigcup_{a \in S} [a]_s$ of active effects of all actions in S is consistent. For actions $a = \langle p, e \rangle$ and atomic effects l of the form x and $\neg x$ (for $x \in X$) define the *effect precondition* $EPC_l(a) = \bigvee \{f \mid f \Rightarrow d \in e, l \in d\}$ where the empty disjunction $\bigvee \emptyset$ is defined as \perp . This formula represents those states in which l is an active effect of a .

Lemma 19.2.1. For literals l , actions a and states s , $l \in [a]_s$ if and only if $s \models EPC_l(a)$.

Let $\pi = \langle X, I, A, G \rangle$ be a *problem instance* consisting of a set X of state variables, a state I over X (the initial state), a set A of actions over X , and a formula G over X (the goal formula).

A (sequential) *plan* for π is a sequence $\sigma = a_1; \dots; a_n$ of actions from A such that $exec_\sigma(I) \models G$. This means that executing the actions in the given order starting in the initial state is defined (the precondition of every action is true and the active effects are consistent when the action is executed) and produces a state that satisfies the goal formula. Sometimes we say that an action sequence is a plan for A and I when we simply want to say that the plan is executable starting from I without specifying the goal states.

19.3. Sequential Plans

Consider a problem instance $\langle X, I, A, G \rangle$. We define, for any $t \geq 0$, the set $X@t = \{x@t \mid x \in X\}$ of propositional variables which contains a variable $x@t$ for every state variable $x \in X$. The variable $x@t$ refers to the value of x in a state

at time t . The propositional variables refer to the values of x at different integer time points. We will later use these superscripts with formulae ϕ so that $\phi@t$ is the formula obtained from ϕ by replacing every $x \in X$ by the corresponding $x@t \in X@t$.

An action $a = \langle p, e \rangle$ can be represented as

$$\tau_a = p@0 \wedge \bigwedge_{x \in X} [(EPC_x(a)@0 \vee (x@0 \wedge \neg EPC_{\neg x}(a)@0)) \leftrightarrow x@1]. \quad (19.1)$$

This formula expresses the precondition of the action and the new value of each state variable x in terms of the values of the state variables before the action is executed: a will be true if it becomes true or it was true already and does not become false.

Lemma 19.3.1. *Let s and s' be states over X . Let $s_0 : X@0 \rightarrow \{0, 1\}$ be a state over $X@0$ such that $s_0(x) = s(x)$ for all $x \in X$. Let $s_1 : X@1 \rightarrow \{0, 1\}$ be a state over $X@1$ such that $s_1(x') = s'(x)$ for all $x \in X$. Then $s_0 \cup s_1 \models \tau_a$ if and only if $s' = \text{exec}_a(s)$.*

The formula for representing the possibility of taking any of the actions is

$$\mathcal{T}(0) = \bigvee_{a \in A} \tau_a. \quad (19.2)$$

Later we will use this formula by replacing occurrences of the propositional variables in $X@0$ and $X@1$ by propositional variables referring to different integer time points.

Lemma 19.3.2. *Let s and s' be states over X . Let $s_0 : X@0 \rightarrow \{0, 1\}$ be a state over $X@0$ such that $s_0(x) = s(x)$ for all $x \in X$. Let $s_1 : X@1 \rightarrow \{0, 1\}$ be a state over $X@1$ such that $s_1(x') = s'(x)$ for all $x \in X$. Then $s_0 \cup s_1 \models \tau_a$ if and only if $s' = \text{exec}_a(s)$. Then $s_0 \cup s_1 \models \mathcal{T}(0)$ if and only if $s' = \text{exec}_a(s)$ for some $a \in A$.*

Since the formula $\mathcal{T}(0)$ expresses the changes in the state of the world between two time points by one action, the changes caused by a sequence of actions over a sequence of states can be expressed by iterating this formula.

Theorem 19.3.3. *Let $\pi = \langle X, I, A, G \rangle$ be a problem instance. Let $\iota = \bigwedge \{x@0 | x \in X | s \models x\} \wedge \bigwedge \{\neg x@0 | x \in X, I \not\models x\}$. The formula*

$$\iota \wedge \mathcal{T}(0) \wedge \mathcal{T}(1) \wedge \cdots \wedge \mathcal{T}(t-1) \wedge G@t \quad (19.3)$$

is satisfiable if and only there is a sequence a_0, a_1, \dots, a_{t-1} of t actions such that $\text{exec}_{a_{t-1}}(\cdots \text{exec}_{a_1}(\text{exec}_{a_0}(I)) \cdots) \models G$.

A satisfying assignment represents the sequence of states that is visited when a plan is executed. The plan can be constructed by identifying actions which correspond to the changes between every pair of consecutive states.

19.3.1. Improvements

This basic representation of planning in the propositional logic can be improved many ways, and effort in implementation technologies for the underlying SAT solving can be critical for the scalability [Rin12b, Rin12a]. Next we briefly discuss two powerful ways to improve the translations of planning into the propositional logic.

19.3.1.1. Approximations of Reachability

Important for the efficiency of planning with SAT is that the search by a SAT algorithm focuses on sequences of states that are reachable from the initial state. Testing whether a state is reachable is as difficult as planning itself but there are efficient algorithms for finding approximate information about reachability, for example, in the form of dependencies between state variables. A typical dependence between some state variables x_1, \dots, x_n is that at most one of them has value 1 at a time, in any reachable state. Adding formulae $\neg x_i \vee \neg x_j$ for all $i \in \{1, \dots, n\}$ and $j \in \{1, \dots, n\}$ such that $i \neq j$ may improve the efficiency of the satisfiability tests tremendously because no states need to be considered that are later found not to be reachable from the initial state. This kind of dependencies are called *invariants*. There are algorithms for finding invariants automatically [Rin98, GS98, Bra01, Rin08]. These algorithms run in polynomial time and find a subset of the invariants that are 2-literal clauses. Invariants were first introduced to planning in the form of *mutexes* in the planning graphs of Blum and Furst [BF97].

19.3.1.2. Factoring

Input languages for planning systems represent sets of actions as an action schema together with objects by which the variables in the action schema are instantiated in order to obtain the non-schematic (ground) actions of Section 19.2. The size of the set of ground actions may be exponential in the size of the schematic representation. Sometimes it is possible to reduce the size increase by representing part of the instantiation process directly as a propositional formula. For example, an action $\text{move}(x, y, z)$ for moving an object x from location y to location z can be represented by the parameters $x \in X$, $y \in Y$ and $z \in Y$ where X is the set of all objects and Y is the set of all locations. Instead of $|X| \times |Y|^2$ propositional variables for representing the ground actions, only $|X| + 2|Y|$ propositional variables are needed for representing the possible parameter values of the move action. This has been called a factored representation [KMS96, EMW97]. Factoring may reduce the potential for parallelism (Section 19.4) in plans: in the move action above, the action should be factored only with respect to the location parameters y and z , as different objects can be moved in parallel, but any given object can be moved only from one location to another. Factoring can speed up the planning process substantially [RGPS09].

19.4. Parallel Plans

An important factor in the efficiency of SAT-based planners is the notion of parallel plans. Few plans are purely sequential in the sense that any two consecutive actions have to be in the given order. For example, the last two actions in a plan for achieving $a \wedge b$ could respectively make a and b true and be completely independent of each other. The ordering of the actions can be reversed without invalidating the plan. This kind of independence naturally leads to the notion of partially ordered plans, which was first utilized in the connection of the so-called non-linear or partial order approach to AI planning [Sac75, MR91]. Techniques utilizing partial orders and independence outside planning include partial-order reduction techniques [God91, Val91, ABH⁺97] for reachability analysis.

The highly influential GraphPlan planner [BF97] introduced a restricted notion of partial ordering which turned out to be very effective for SAT-based planning [KS96]: a plan is viewed as a sequence of sets of actions.

In this section we discuss two forms of partially ordered plans and present their translations into the propositional logic. The first definition generalizes that of Blum and Furst [BF97]. The second was proposed by Dimopoulos et al. [DNK97] and later formalized and applied to SAT-based planning [RHN06].

19.4.1. \forall -Step Plans

The first definition of parallel plans interprets parallelism as the possibility of ordering the actions to any total order.

Definition 19.4.1 (\forall -Step plans). For a set of actions A and an initial state I , a \forall -step plan for A and I is a sequence $T = \langle S_0, \dots, S_{l-1} \rangle$ of sets of actions for some $l \geq 0$ such that there is a sequence of states s_0, \dots, s_l (the execution of T) such that

1. $s_0 = I$, and
2. $exec_{a_1, \dots, a_n}(s_i)$ is defined and equals s_{i+1} for every $i \in \{0, \dots, l-1\}$ and every total ordering a_1, \dots, a_n of S_i .

When active effects are independent of the current state and preconditions are conjunctions of literals, Definition 19.4.1 exactly corresponds to a syntactic definition of independence [RHN06]. In more general cases the correspondence breaks down, and syntactically independent sets of actions are only a subclass of sets of actions which can be ordered to any total order.

Testing whether a sequence of sets of actions is a \forall -step plan is in general intractable [RHN06] which justifies syntactic rather than semantic restrictions on parallel actions [BF97, KS96].

We define positive and negative occurrences of state variables in a formula.

Definition 19.4.2 (Positive and negative occurrences). We say that a state variable a occurs positively in ϕ if $positive(a, \phi)$ is true. Similarly, a occurs negatively

in ϕ if $\text{negative}(a, \phi)$ is true.

$$\begin{aligned} \text{positive}(x, x) &= \text{true, for all } x \in X \\ \text{positive}(x, y) &= \text{false, for all } \{x, y\} \subseteq X \text{ such that } x \neq y \\ \text{positive}(x, \phi \vee \phi') &= \text{positive}(x, \phi) \text{ or } \text{positive}(x, \phi') \\ \text{positive}(x, \neg\phi) &= \text{negative}(x, \phi) \end{aligned}$$

$$\begin{aligned} \text{negative}(x, y) &= \text{false, for all } \{x, y\} \subseteq X \\ \text{negative}(x, \phi \vee \phi') &= \text{negative}(x, \phi) \text{ or } \text{negative}(x, \phi') \\ \text{negative}(x, \neg\phi) &= \text{positive}(x, \phi) \end{aligned}$$

A state variable x occurs in ϕ if it occurs positively or negatively in ϕ .

Definition 19.4.3 (Affect). Let X be a set of state variables and $a = \langle p, e \rangle$ and $a' = \langle p', e' \rangle$ actions over X . Then a affects a' if there is $x \in X$ such that

1. $x \in d$ for some $f \Rightarrow d \in c$ and x occurs in f for some $f \Rightarrow d \in c'$ or occurs negatively in p' , or
2. $\neg x \in d$ for some $f \Rightarrow d \in c$ and x occurs in f for some $f \Rightarrow d \in c'$ or occurs positively in p' .

An action a which affects another action a' may prevent its execution or change its active effects. This means that replacing $a'; a$ by $a; a'$ in a plan may make the plan invalid or change its execution.

Definition 19.4.4 (Interference). Let X be a set of state variables. Actions a and a' interfere if a affects a' or a' affects a .

If actions a and a' do not interfere, then the sequences $a; a'$ and $a'; a$ are interchangeable. Non-interference is a sufficient but not a necessary condition for interchangeability.

19.4.2. \exists -Step Plans

A more relaxed notion of parallel plans was proposed by Dimopoulos et al. [DNK97] and formalized and investigated in detail by Rintanen et al. [RHN06].

Definition 19.4.5 (\exists -Step plans). For a set A of actions and an initial state I , a \exists -step plan is $T = \langle S_0, \dots, S_{l-1} \rangle \in (2^A)^l$ together with a sequence of states s_0, \dots, s_l (the execution of T) for some $l \geq 0$ such that

1. $s_0 = I$, and
2. for every $i \in \{0, \dots, l-1\}$ there is a total ordering $a_1 < \dots < a_n$ of S_i such that $s_{i+1} = \text{exec}_{a_1; \dots; a_n}(s_i)$.

The difference to \forall -step plans is that instead of requiring that each step S_i can be ordered to any total order, it is sufficient that there is at least one order that maps state s_i to s_{i+1} . Unlike for \forall -step plans, the successor s_{i+1} of s_i is not uniquely determined solely by S_i because the successor depends on the implicit ordering of S_i . For this reason the definition has to make the execution s_0, \dots, s_l explicit.

The more relaxed definition of \exists -step plans sometimes allows much more parallelism than the definition of \forall -step plans.

Example 19.4.1. Consider a row of n Russian dolls, each slightly bigger than the preceding one. We can nest all the dolls by putting the first inside the second, then the second inside the third, and so on, until every doll except the largest one is inside another doll.

For four dolls this can be formalized as follows.

$$\begin{aligned} a_1 &= \langle \text{out1} \wedge \text{out2} \wedge \text{empty2}, \{\top \Rightarrow \text{1in2}, \top \Rightarrow \neg \text{out1}, \top \Rightarrow \neg \text{empty2}\} \rangle \\ a_2 &= \langle \text{out2} \wedge \text{out3} \wedge \text{empty3}, \{\top \Rightarrow \text{2in3}, \top \Rightarrow \neg \text{out2}, \top \Rightarrow \neg \text{empty3}\} \rangle \\ a_3 &= \langle \text{out3} \wedge \text{out4} \wedge \text{empty4}, \{\top \Rightarrow \text{3in4}, \top \Rightarrow \neg \text{out3}, \top \Rightarrow \neg \text{empty4}\} \rangle \end{aligned}$$

The shortest \forall -step plan nests the dolls in three steps: $\langle \{a_1\}, \{a_2\}, \{a_3\} \rangle$. The \exists -step plan $\langle \{a_1, a_2, a_3\} \rangle$ nests the dolls in one step.

Theorem 19.4.1. (i) Every \forall -step plan is a \exists -step plan, and (ii) for every \exists -step plan T there is a \forall -step plan whose execution leads to the same final state as that of T .

Similarly to \forall -step plans, testing the validity of \exists -step plans is intractable in the worst case [RHN06]. One way of achieving tractability uses two restrictions. All parallel actions are required to be executable in the current state (unlike for \forall -step plans this does not follow from the definition), and parallel actions are required to fulfil an ordered dependence condition based on Definition 19.4.3: actions are allowed in parallel if they can be totally ordered so that no action affects a later action. This means that the actions preceding a given action do not change its effects or prevent its execution.

Theorem 19.4.2. Let A be a set of actions, I a state, $T = \langle S_0, \dots, S_{l-1} \rangle \in (2^A)^l$, and s_0, \dots, s_l a sequence of states. If

1. $s_0 = I$,
2. for every $i \in \{0, \dots, l-1\}$ there is a total ordering $<$ of S_i such that if $a < a'$ then a does not affect a' , and
3. $s_{i+1} = \text{exec}_{S_i}(s_i)$ for every $i \in \{0, \dots, l-1\}$,

then T is a \exists -step plan for A and I .

Even though the class of \exists -step plans based on *affects* is narrower than the class sanctioned by Definition 19.4.5, much more parallelism is still possible in comparison to \forall -step plans. For instance, nesting of Russian dolls in Example 19.4.1 belongs to this class.

19.4.3. Optimality of Parallel Plans

The most commonly used measure for plan quality is the number of actions in the plan. So the problem of finding an optimal plan is the problem of finding the shortest sequential plan, or in terms of the satisfiability tests, finding an integer t such that the formula for plan length $t-1$ is unsatisfiable and the formula for plan length t is satisfiable. For parallel plans the question is more complicated. It is possible to find an optimal parallel plan for a given parallel length by using cardinality constraints on the number of actions in the plans, but it seems that

finding an optimal parallel plan in the worst case reduces to testing the existence of sequential plans [BR05]: if there is a parallel plan with t time points and n actions, the optimality test may require testing whether there is a sequential plan with $n - 1$ time points and $n - 1$ actions. Tests restricting to formulae with t time points are not sufficient.

19.4.4. Representation in SAT

In this section we present translations of parallel plans into the propositional logic. A basic assumption is that for sets S of simultaneous actions executed in state s the state $exec_S(s)$ is defined, that is, all the preconditions are true in s and the set of active effects of the actions is consistent.

19.4.4.1. The Base Translation

Planning is performed by propositional satisfiability testing by producing formulae $\phi_0, \phi_1, \phi_2, \dots$ such that ϕ_l is satisfiable iff there is a plan of length l . The translations for different forms of parallel plans differ only in the formulae that restrict the simultaneous execution of actions. Next we describe the part of the formulae that is shared by both definitions of parallel plans.

Consider the problem instance $\pi = \langle X, I, A, G \rangle$. Similarly to the state variables in X , for every action $a \in A$ we have the propositional variable a^t for expressing whether a is executed at time point $t \in \{0, \dots, l - 1\}$.

A formula $\Phi_{\pi, l}$ is generated to answer the following question. Is there an execution of a sequence of l sets of actions that reaches a state satisfying G when starting from the state I ? The formula $\Phi_{\pi, l}$ is conjunction of $\iota = \bigwedge \{x@0 \mid x \in X, s \models x\} \wedge \bigwedge \{\neg x@0 \mid x \in X, I \not\models x\}, G@l$, and the formulae described below, instantiated with all $t \in \{0, \dots, l - 1\}$.

First, for every $a = \langle p, e \rangle \in A$ there are the following formulae. The precondition p has to be true when the action is executed.

$$a@t \rightarrow p@t \tag{19.4}$$

Then we define, for every $x \in X$, the condition under which x becomes true or false at step t

$$causes(x)@t = \bigvee_{a \in A} (a@(t-1) \wedge EPC_x(a)@(t-1)) \tag{19.5}$$

$$causes(\neg x)@t = \bigvee_{a \in A} (a@(t-1) \wedge EPC_{\neg x}(a)@(t-1)) \tag{19.6}$$

These formulas are used in defining when a state variable becomes true or false at a given time point.

$$causes(x)@t \rightarrow x@t \tag{19.7}$$

$$causes(\neg x)@t \rightarrow \neg x@t \tag{19.8}$$

Second, the value of a state variable does not change if no action that changes it is executed. Hence for every state variable x we have two formulae, one expressing the conditions for the change of x from true to false,

$$(x@t \wedge \neg x@(t+1)) \rightarrow \text{causes}(\neg x)@(t+1) \quad (19.9)$$

and another from false to true,

$$(\neg x@t \wedge x@(t+1)) \rightarrow \text{causes}(x)@(t+1). \quad (19.10)$$

The formulae $\Phi_{\pi,l}$, just like the definition of $\text{exec}_S(s)$, allow sets of actions in parallel that do not correspond to any sequential plan. For example, the actions $\langle x, \{\top \Rightarrow \neg y\} \rangle$ and $\langle y, \{\top \Rightarrow \neg x\} \rangle$ may be executed simultaneously resulting in a state satisfying $\neg x \wedge \neg y$, even though this state is not reachable by the two actions sequentially. But we require that all parallel plans can be executed sequentially. Further formulae that are discussed in the next sections are needed to guarantee this property.

Theorem 19.4.3. *Let $\pi = \langle X, I, A, G \rangle$ be a transition system. Then there is $T = \langle S_0, \dots, S_{l-1} \rangle \in (2^A)^l$ so that s_0, \dots, s_l are states so that $I = s_0, s_l \models G$, and $s_{i+1} = \text{exec}_{S_i}(s_i)$ for all $i \in \{0, \dots, l-1\}$ if and only if there is a valuation satisfying the formula $\Phi_{\pi,l}$.*

Proposition 19.4.4. *The size of the formula $\Phi_{\pi,l}$ is linear in l and in the size of π .*

Theorem 19.4.3 says that a sequence of actions fulfilling certain conditions exists if and only if a given formula is satisfiable. The theorems connecting certain formulae to certain notions of plans (Theorems 19.4.5 and 19.4.6) provide an implication only in one direction: whenever the formula for a given value of parameter l is satisfiable, a plan of l time points exists. The other direction is missing because the formulae in general only approximate the respective definition of plans and there is no guarantee that the formula for a given l is satisfiable when a plan with l time points exists. However, the formula with some higher value of l is satisfiable. This follows from the fact that whenever a \forall -step or \exists -step plan $\langle S_0, \dots, S_{l-1} \rangle$ with $n = |S_0| + \dots + |S_{l-1}|$ occurrences of actions exists, there is a plan consisting of n singleton sets, and the corresponding formulae $\Phi_{\pi,n} \wedge \Phi_{A,n}^x$ are satisfiable. The formulae $\Phi_{A,n}^z$ represent the parallel semantics z for actions A .

19.4.4.2. Translation of \forall -Step Plans

The simplest representation of the interference condition in Definition 19.4.4 is by formulae

$$\neg(a_1@t \wedge a_2@t) \quad (19.11)$$

for every pair of interfering actions a_1 and a_2 . Define $\Phi_{A,l}^{\forall\text{step}}$ as the conjunction of the formulae (19.11) for all time points $t \in \{0, \dots, l-1\}$ and for all pairs of interfering actions $\{a, a'\} \subseteq A$ that could be executed simultaneously. There are $O(ln^2)$ such formulae for n actions. This can be improved to linear [RHN06].

Theorem 19.4.5. *Let $\pi = \langle X, I, A, G \rangle$ be a transition system. There is a \forall -step plan of length l for π if $\Phi_{\pi,l} \wedge \Phi_{A,l}^{\forall\text{step}}$ is satisfiable.*

19.4.4.3. Translation of \exists -Step Plans

The simplest efficient representation of \exists -step plans imposes a fixed total ordering on the actions and allows the simultaneous execution of a subset of the actions only if none of the actions affects any action later in the ordering. There are more permissive ways of guaranteeing the validity of \exists -step plans, but they seem to be difficult to implement efficiently, and furthermore, there are ways of identifying most permissive total orderings as a preprocessing step by finding strongly connected components of the graph formed by the *affects* relation on actions [RHN06].

The representation does not allow all the possible parallelism but it leads to small formulae and is efficient in practice. In the translation for \exists -step plans the set of formulae constraining parallel execution *is a subset* of those for the less permissive \forall -step plans. One therefore receives two benefits simultaneously: possibly much shorter parallel plans and formulae with a smaller size / time points ratio.

The idea is to impose beforehand an (arbitrary) ordering on the actions a_1, \dots, a_n and to allow parallel execution of two actions a_i and a_j such that a_i affects a_j only if $i \geq j$. The formula $\Phi_{A,l}^{\exists\text{step}}$ is the conjunction of

$$\neg(a_i@t \wedge a_j@t) \text{ if } a_i \text{ affects } a_j \text{ and } i < j$$

for all a_i and a_j and all time points $t \in \{0, \dots, l-1\}$. Of course, this restriction to one fixed ordering may rule out many sets of parallel actions that could be executed simultaneously according to some other ordering than the fixed one.

The formula $\Phi_{A,l}^{\exists\text{step}}$ (similar to the translation for \forall -step plans in Section 19.4.4.2) has a quadratic size because of the worst-case quadratic number of pairs of actions that may not be simultaneously executed. This can be improved to linear [RHN06].

Theorem 19.4.6. *Let $\pi = \langle X, I, A, G \rangle$ be a transition system. There is a \exists -step plan of length l for π if $\Phi_{\pi,l} \wedge \Phi_{A,l}^{\exists\text{step}}$ is satisfiable.*

19.5. Finding a Satisfiable Formula

Given a sequence ϕ_0, ϕ_1, \dots of formulae representing the bounded planning problem for different plan lengths, a satisfiability algorithm is used for locating a satisfiable formula. The satisfying assignment can be translated into a plan. In this section we describe three high-level algorithms that use the formulae ϕ_i for finding a plan. We call them Algorithms S, A and B.

Algorithm S is the obvious sequential procedure for finding a satisfiable formula: first test the satisfiability of ϕ_0 , then ϕ_1 , ϕ_2 , and so on, until a satisfiable formula is found. It has been used by virtually all works that reduce planning to the fixed-length planning problem [KS92, BF97]. This algorithm has the property that it always find a plan with the smallest number of time points. It can be used for finding plans with the minimum number of actions if used in connection with the sequential encoding of planning from Section 19.3.

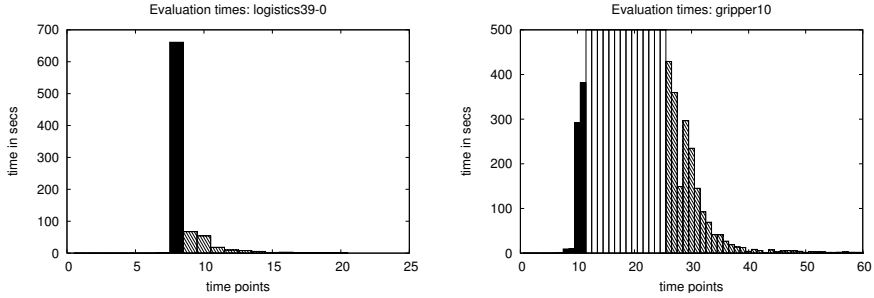


Figure 19.1. SAT solver runtimes for two problem instances and different plan lengths

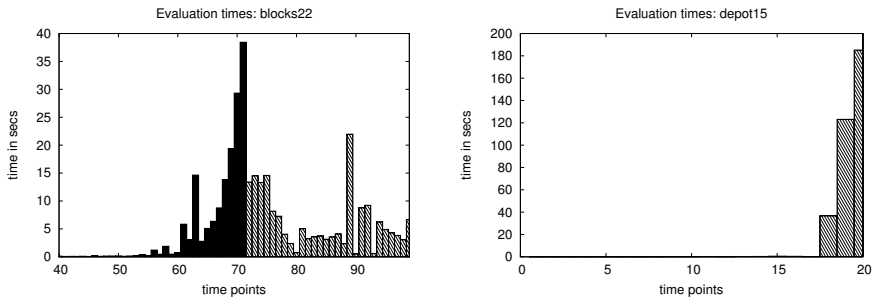


Figure 19.2. SAT solver runtimes for two problem instances and different plan lengths

If the objective is to find a plan with not necessarily the smallest number of actions or time points, the Algorithm S is often inefficient because the satisfiability tests for the last unsatisfiable formulae are often much more expensive than for the first satisfiable formulae. Consider the diagrams in Figures 19.1, 19.2 and Figure 19.3. Each diagram shows the cost of detecting the satisfiability or unsatisfiability of formulae that represent the existence of plans of different lengths. Grey bars depict unsatisfiable formulae and black bars satisfiable formulae. For more difficult problems the peak formed by the last unsatisfiable formulae is still much more pronounced.

Except for the rightmost diagram in Figure 19.2 and the leftmost diagram in Figure 19.3, the diagrams depict steeply growing costs of determining unsatisfiability of a sequence of formulae followed by small costs of determining satisfiability of formulae corresponding to plans.

We would like to run a satisfiability algorithm with the satisfiable formula for which the runtime of the algorithm is the lowest. Of course, we do not know which formulae are satisfiable and which has the lowest runtime. With an infinite number of processes we could find a satisfying assignment for one of the formulae in the smallest possible time: let each process $i \in \{0, 1, 2, \dots\}$ test the satisfiability of ϕ_i . However, an infinite number of processes running at the same speed cannot be simulated by any finite hardware. Algorithms A and B attempt to approximate

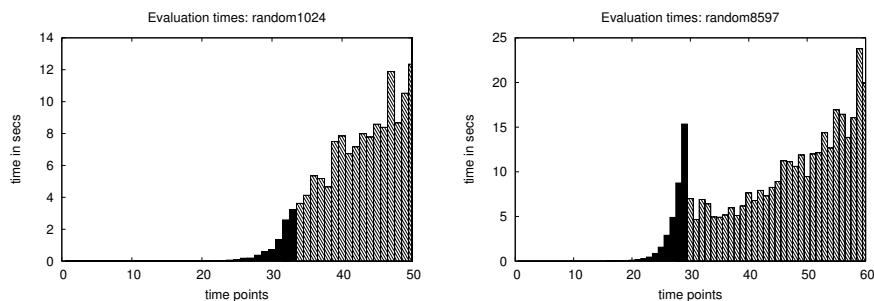


Figure 19.3. SAT solver runtimes for two problem instances and different plan lengths

```

1: procedure AlgorithmS()
2:    $i := 0$ ;
3:   repeat
4:     test satisfiability of  $\phi_i$ ;
5:     if  $\phi_i$  is satisfiable then terminate;
6:      $i := i + 1$ ;
7:   until 1=0;

```

Figure 19.4. Algorithm S

this scheme by using a finite number processes. Algorithm A uses a fixed number n of processes. Algorithm B uses a finite but unbounded number of processes which are run at variable rates.

19.5.1. Algorithm S: Sequential Testing

The standard algorithm for finding plans in the satisfiability and related approaches to planning tests the satisfiability of formulae for plan lengths 0, 1, 2, and so on, until a satisfiable formula is found [BF97, KS96]. This algorithm is given in Figure 19.4.

19.5.2. Algorithm A: Multiple Processes

Algorithm A (Figure 19.5) distributes plan search to n concurrent processes and initially assigns the first n formulae to the n processes. Whenever a process finds its formula satisfiable, the computation is terminated. Whenever a process finds its formula unsatisfiable, the process is given the next unassigned formula. This strategy can avoid completing the satisfiability tests of many of the expensive unsatisfiable formulae, thereby saving a lot of computation effort.

Algorithm S is the special case with $n = 1$. The constant ϵ determines the coarseness of CPU time division. The *for each* loop in this algorithm and in the next algorithm can be implemented so that several processes are run in parallel.

```

1: procedure AlgorithmA( $n$ )
2:    $P := \{\phi_0, \dots, \phi_{n-1}\}$ ;
3:    $\text{next} := n$ ;
4:   repeat
5:      $P' := P$ ;
6:     for each  $\phi \in P'$  do
7:       continue computation with  $\phi$  for  $\epsilon$  seconds;
8:       if  $\phi$  was found satisfiable then terminate;
9:       if  $\phi$  was found unsatisfiable then
10:          $P := P \cup \{\phi_{\text{next}}\} \setminus \{\phi\}$ ;
11:          $\text{next} := \text{next} + 1$ ;
12:       end if
13:     end do
14:   until  $0=1$ 

```

Figure 19.5. Algorithm A

```

1: procedure AlgorithmB( $\gamma$ )
2:    $t := 0$ ;
3:   for each  $i \geq 0$  do  $\text{done}[i] = \text{false}$ ;
4:   for each  $i \geq 0$  do  $\text{time}[i] = 0$ ;
5:   repeat
6:      $t := t + \delta$ ;
7:     for each  $i \geq 0$  such that  $\text{done}[i] = \text{false}$  do
8:       if  $\text{time}[i] + n\epsilon \leq t\gamma^i$  for some maximal  $n \geq 1$  then
9:         continue computation for  $\phi_i$  for  $n\epsilon$  seconds;
10:        if  $\phi_i$  was found satisfiable then terminate;
11:         $\text{time}[i] := \text{time}[i] + n\epsilon$ ;
12:        if  $\phi_i$  was found unsatisfiable then  $\text{done}[i] := \text{true}$ ; end if
13:      end if
14:    end do
15:   until  $0=1$ 

```

Figure 19.6. Algorithm B

19.5.3. Algorithm B: Geometric Division of CPU Use

Algorithm B (Figure 19.6) uses an unbounded but finite number of processes. The amount of CPU given to each process depends on the index of its formula: if formula ϕ_k is given t seconds of CPU during a certain time interval, then a formula ϕ_i , $i \geq k$ is given $\gamma^{i-k}t$ seconds. This means that every formula gets only slightly less CPU than its predecessor, and the choice of the exact value of the constant $\gamma \in]0, 1[$ is less critical than the choice of n for Algorithm A.

Variable t , which is repeatedly increased by δ , characterizes the total CPU time $\frac{t}{1-\gamma}$ available so far. As the computation for ϕ_i proceeds only if it has been going on for at most $t\gamma^i - \epsilon$ seconds, CPU is actually consumed less than $\frac{t}{1-\gamma}$, and there will be at time $\frac{t}{1-\gamma}$ only a finite number $j \leq \log_\gamma \frac{\epsilon}{t}$ of formulae for which computation has commenced.

The constants n and γ respectively for Algorithms A and B are roughly related by $\gamma = 1 - \frac{1}{n}$: of the CPU capacity $\frac{1}{n} = 1 - \gamma$ is spent in testing the first unfinished formula. Algorithm S is the limit of Algorithm B when γ goes to 0.

Algorithms A and B have two very useful properties [RHN06]. First, both al-

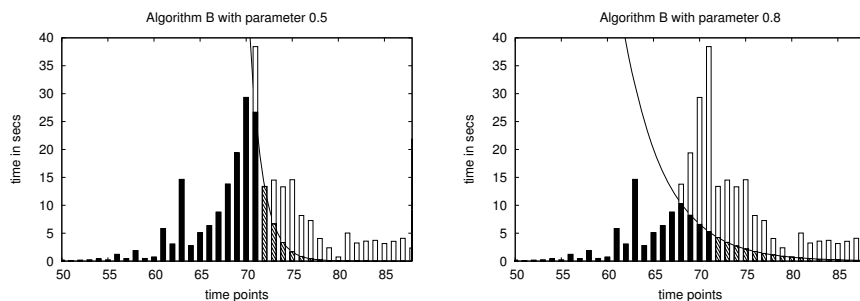


Figure 19.7. Illustration of two runs of Algorithm B. When $\gamma = 0.5$ most CPU time is spent on the first formulae, and when the first satisfiable formula is detected also the unsatisfiability of most of the preceding unsatisfiable formulae has been detected. With $\gamma = 0.8$ more CPU is spent for the later easier satisfiable formulae, and the expensive unsatisfiability tests have not been completed before finding the first satisfying assignment.

gorithms can be arbitrarily much faster than Algorithm S. This is because testing the satisfiability of some of the unsatisfiable formulae may be arbitrarily difficult in comparison to the easiest satisfiable formulae. Second, both algorithms are at most a constant factor slower than Algorithm S. This constant is n for Algorithm A and $\frac{1}{1-\gamma}$ for Algorithm B. So even in cases in which these algorithms do not improve on Algorithm S, the efficiency loss is bounded.

19.6. Temporal Planning

The SAT modulo Theories (SMT) [WW99, ABC⁺02] framework, which supports numerical and other constraints in addition to Boolean ones, can be applied to the solution of more general planning problems.

The first application of SAT extended with numerical constraints was by Wolfman and Weld [WW99] to the classical planning problem with numerical state variables. The notion of plans and executions are the same as in classical planning. The difference is that in addition Boolean state variables, there are numerical state variables which can be assigned values expressed as arithmetic expressions.

Shin and Davis gave a systematic translation of an expressive planning language into the SMT framework [SD05]. This is a major generalization of the representations of classical planning as propositional formulas. Instead of a sequence of states, with a number of actions taking between any two consecutive states, in temporal planning there can be multiple actions temporally overlapping each other in different ways, and the execution of one action can span multiple steps of the plan. This framework can be applied to a range of modelling languages [Rin17], and as the number of steps of a plan is bounded similarly to classical planning, the same solving strategies (Section 19.5) are applicable.

Here we sketch the main ideas of the type of encodings first proposed by Shin and Davis [SD05] which have later been used for solving planning problems by using SMT solvers.

We use a simple formalization of temporal planning, with a set X of state variables and actions as triples (p, d, e) , where p is a propositional formula over X just like in classical planning, and $d > 0$ is a rational *duration* of the action, and e is a set of literals over X made true by the action. The precondition p has to be true at the time point in which the action is taken, and the effects e become true after time d .

Unlike in classical planning, timelines in temporal planning are continuous, either rational or real valued, but it is still sufficient to represent explicitly only a finite sequence of time points in which an action is taken or an effect of an action takes place, just like in classical planning.

As before, l denotes the index of the last step so that the $l + 1$ steps of a plan are indexed from 0 to l . We use the same atomic propositions $a@i$ and $x@i$ to indicate whether an action is taken or a state variable is true at a given step i . Additionally, we associate an absolute time $\tau@i$ with each step. The numeric inequalities

$$\tau@i > \tau@j \quad (19.12)$$

for all i and j such that $l \geq j > i > 0$, forces the absolute time to be monotonically increasing. Next we express the properties of actions and state variables at different steps of a plan.

We have the formula

$$a@i \rightarrow p@i \quad (19.13)$$

for the precondition p of action a , for every $i \in \{0, \dots, l - 1\}$.

For an action $a = (p, d, e)$, define $EPC_i(a) = \top$ iff $l \in e$ and otherwise $EPC_i(a) = \perp$. Like earlier, we define $causes(l)@i$ to denote the conditions under which literal l becomes true at step i . For temporal planning this can refer to multiple earlier steps, as the last step might not be the one in which an action affecting l took place.

$$causes(x)@i = \bigvee_{a \in A} \bigvee_{j=0}^{i-1} (a@j \wedge EPC_x(a)@j \wedge (\tau@i - \tau@j = d_a)) \quad (19.14)$$

we denote the disjunction of all the conditions under which x becomes true at step $i \in \{0, \dots, l\}$. These formulas refer to atomic propositions for earlier steps. Similarly $causes(\neg x)@i$ for x becoming false. Hence when $causes(l)@i$ is true, the effect l takes place.

$$causes(x)@i \rightarrow x@i \quad (19.15)$$

$$causes(\neg x)@i \rightarrow \neg x@i \quad (19.16)$$

Frame axioms allow inferring that the value of a state variable remains unchanged.

$$(x@i \wedge \neg x@(i - 1)) \rightarrow causes(x)@i \quad (19.17)$$

$$(\neg x@i \wedge x@(i - 1)) \rightarrow causes(\neg x)@i \quad (19.18)$$

When $t > 0$, to guarantee that there is a step for every effect, we need

$$a@i \rightarrow \bigvee_{j=i+1}^l (\tau@j - \tau@i = t). \quad (19.19)$$

Most temporal planning problems restrict the possibilities to overlap two actions in various ways. Assume that no overlap for actions a and a' is allowed, and these actions respectively have durations d and d' . Now, if a is taken at t , then a' cannot be started before time $t + d$, and if a' is taken at t , then a cannot be started before time $t + d'$, expressed by

$$a@i \rightarrow \bigwedge_{j=i}^l (a'@j \rightarrow (\tau@j - \tau@i \geq d)) \quad (19.20)$$

$$a'@i \rightarrow \bigwedge_{j=i}^l (a@j \rightarrow (\tau@j - \tau@i \geq d')) \quad (19.21)$$

For different limits on the concurrency of actions see the literature [SD05, Rin17].

Temporal planning can be reduced to plain SAT [RGS15], but as numerical durations are not represented, a satisfying assignment is not guaranteed to correspond to a valid plan, and multiple calls, with tighter Boolean constraints, may be required to find one. Also, minimization of the durations of plans is harder.

19.7. Contingent Planning

Classical planning and temporal planning assume the world to be completely predictable. Nondeterminism and unpredictability, as well as limited observability, complicate planning and reduce the possibilities of effective reductions to SAT.

With a polynomial bound on the number of actions in a plan, the complexity of classical planning decreases from PSPACE-complete to NP-complete. More general planning problems have a complexity far exceeding PSPACE: planning with nondeterministic actions and full observability is EXP-complete [Lit97], with no observability EXPSPACE-complete [HJ00], and with partial observability 2-EXP-complete [Rin04]. Merely bounding plans to polynomial size will not decrease the complexity to NP. There are two options: either use a formalism harder than NP, or impose further restrictions. Both options have been considered.

In planning with full or partial observability, a plan may have to handle different contingencies in order to achieve its objectives (which is why these problems are sometimes called contingent planning.) Plans can no longer be limited to sequences of actions, as different observations made during the execution may lead to alternative courses of action, represented for example as a branching plan, or a mapping from observations (and the execution state) to actions.

If both plan sizes and executions are restricted to be polynomial, the planning problem is in Σ_2^P [Rin99]. Problems belonging to this complexity class correspond to quantified Boolean formulas (QBF) [SM73] with the prefix $\exists\forall$ (Section 19.7.1).

Even stricter restrictions can reduce the complexity to NP. Chatterjee et al. [CCD16] have shown that planning with partial observability, when the state space is represented simply as a graph with states as nodes, is NP-complete, and can therefore be reduced to SAT (Section 19.7.2).

For problems with very large state spaces, the explicit representation of all states is not feasible. Geffner and Geffner [GG18] have shown that, for a relatively broad class of contingent planning problems with full observability and actions

and states represented in terms of state variables, reduction to SAT can still be a competitive solution method (Section 19.7.3).

19.7.1. Contingent Planning in Σ_2^P with QBF

In this section we discuss the use of quantified Boolean formulas for solving simple contingent planning problems with a polynomial length bound, and more expressive contingent planning problems with a size bound and other, more restrictive conditions to bring the complexity down to NP.

The complexity class Σ_2^P corresponds to QBF with the prefix $\exists\forall$. However, the most natural translation of the planning problem into QBF has the prefix $\exists\forall\exists$, corresponding to problems in Σ_3^P . This discrepancy can be interpreted as the power of QBF with prefix $\exists\forall\exists$ to handle a generalized planning problem in which the possibility of taking an action in a given state and computing the unique successor state cannot be performed in polynomial time [Tur02]. In this section we present the most obvious translation that uses the prefix $\exists\forall\exists$. Translation with prefix $\exists\forall$ is less intuitive [Rin07]. The planning problem with several initial states is similar to the problem of finding homing or reset or synchronizing sequences of transition systems [PB91].

The uncertainty about the initial state as well as the nondeterminism of actions and the environment can be represented as state variables that can assume arbitrary values and different ones at different steps of the executions. We call these state variables the *contingency variables*.

Actions cannot change the values of the contingency variables, but the actions can otherwise depend on them: contingency variables can occur in the actions' preconditions and the condition parts of the effects, thereby affecting the applicability of actions and the effects of actions.

We define a contingent planning problem as a tuple $\langle X, C, I, A, G \rangle$, where X is the set of all state variables, $C \subseteq X$ is the contingency variables, I is a formula for the initial states, A is the set of actions, and G is the formula for the goal states. Here X , A , and G are like for classical in Section 19.2. The contingency variables and the definition of I as a formula are where these definitions diverge.

Next we define a quantified Boolean formula that represents the problem of finding a sequence of actions that leads from any of the initial state to any of the goal states, under any possible combination of contingencies.

Define $P = \{a@i | a \in A, 0 \leq i \leq l\}$ for the set of atomic propositions representing the action and thereby the plan. Define $Z = \{x@i | x \in C, 0 \leq i \leq l - 1\} \cup \{x@0 | x \in X \setminus C\}$ for the values of the contingency variables at all points of the execution, and also the values of all other state variables at the first step of the execution (the initial state). Define $E = \{x@i | x \in X, 1 \leq i \leq l\}$ for the values of non-contingency state variables at all steps of the execution after the initial state. The values of these state variables are determined unique by the action variables and the values of the contingency variables on the previous steps.

Now the translation of the planning problem into QBF is as follows.

$$\exists P \forall Z \exists E (I@0 \rightarrow (\mathcal{T}(0) \wedge \cdots \wedge \mathcal{T}(l-1) \wedge G@l)) \quad (19.22)$$

The formula says that there is a plan (expressed by P) such that for all combinations of contingencies (expressed by Z), there is an execution (expressed by E) that leads to a state that satisfies the goal formula G .

This QBF evaluates to *true* if and only if such a plan exists. For a true QBF the valuation of the outermost variables P indicates what the plan is.

The above formula representing planning *without* observability: a single sequence of actions has to reach the goals independently of the contingencies. In more general cases, observations made during the execution can determine how the plan execution is continued. This corresponds to plans with branching, and the variables P above could represent such a plan, together with a more complex body for the QBF to express how such a branching plan would determine which actions are taken during the execution [Rin99].

The QBF-based representation can be generalized also in other ways. A probabilistic extension of QBF, stochastic satisfiability or SSAT, makes it possible to express transition probabilities associated with nondeterministic actions and noisy observations [ML03].

19.7.2. Planning with Partial Observability in NP

Chatterjee et al. [CCD16] observe that contingent planning with partial observability is in NP if limiting the memory available during the execution, and considering an enumerative – rather than a state variable based – representation of the state space. A graph represents all possible executions of all possible plans. A given plan selects a subset of the arcs of the graph, and the plan must be chosen so that the goals are reached on all execution of the given plan.

Despite the explicit representation of the state space as a graph, the problem is still NP-hard because the actions (outgoing arcs) for each state cannot be chosen independently of other states, and, instead, all states for which the observation (and possibly memory) are the same, must have the same action. This problem of choosing a suitable action for every observation so that goals are reached, makes the problem harder than the corresponding fully observable problem which could be solved in polynomial time.

First we formalize the planning problem.

Definition 19.7.1. A transition system $P = (S, A, O, \delta, Z, I, G)$ consists of

- S is a finite set of states,
- A is a finite set of actions,
- O is a finite set of observations,
- $\delta \subseteq S \times A \times S$ is the transition relation,
- $Z : S \rightarrow O$ is the observation function,
- $I \subseteq S$ is the set of initial states, and
- $G \subseteq S$ is the set of goal states.

The transition relation $\delta(s, a, s')$ expresses whether a transition from s to s' is possible with action a . The function Z associates an observation $Z(s) \in O$ with every state s .

Plans associate an action with every observation. Executions under a given plan include an arc from s to s' if the observation $Z(s)$ in s is associated with

the action a , and $(s, a, s') \in \delta$. A valid plan is one that guarantees that for any state reachable from an initial state, it is always possible to reach one of the goal states.

We will next describe the reduction of this problem to SAT. A core part of it is – similarly to the problem considered in Section 19.7.3 – the representation of constraints expressing the constraint that a given node is (or is not) reachable from another node. For simplicity, we only correspond the memory-free case: generalization to multiple memory states is obtained with a straightforward extension of the formulas given here [CCD16].

The encodings rely on implicitly representing all possible plans and all possible execution graphs. The choice of a plan – the choice of actions and memory updates for all observation and memory state combinations – induces one particular execution graph. A plan is determined by a truth-value assignment to variables π below, and the remaining variables represent the corresponding execution graph (through the variables Arc) and its properties.

A number of propositional variables are defined to encode contingent planning problem with bounded memory into SAT. The first two represent the functions π of bounded-memory plans.

- $\pi(o, a)$ means that action a is taken on observation o .
- $Arc(s, s')$ means that there is a transition from state s to state s' , for all $(s, s') \in Y$ where $Y = \{(s, s') \in S \times S \mid (s, a, s') \in \delta \text{ for some } a \in A\}$.
- $R^I(s)$ means that s is reachable from an initial node under the current plan.
- $R^G(s)$ means that a goal node is reachable from s under the current plan.

The arc variables determine whether a given arc (s, s') is considered to be present in the graph representing all executions of a given plan expressed by the valuation of the variables $\pi(o, a)$.

The value of $\pi(o, a)$ has to be unique for all o and a :

$$\neg(\pi(o, a_1) \wedge \pi(o, a_2)) \quad (19.23)$$

for all $o \in O, a_1 \in A, a_2 \in A$ such that $a_1 \neq a_2$.

The function π has to be defined for states that are reachable: for all $s \in S$, if s is reachable from an initial state, then at least one of the actions executable in s should be enabled in s and hence part of the plan.

$$R^I(s) \rightarrow \bigvee_{a \in X(s)} \pi(Z(s), a) \quad (19.24)$$

Here, $X(s) = \{a \in A \mid (s, a, s') \in \delta, \text{ for some } s'\}$. Whether an arc is included in the graph of possible executions is determined as follows.

$$Arc(s, s') \leftrightarrow \left(\bigvee_{a \in Y(s, s')} \pi(Z(s), a) \right) \quad (19.25)$$

Here, $Y(s, s') = \{a \in A \mid (s, a, s') \in \delta\}$. Finally, it is required that a goal state is reachable from any state that is reachable from an initial state.

$$R^I(s) \rightarrow R^G(s) \quad (19.26)$$

There are alternative ways of encoding these reachability constraints, including a set of clauses and using a specialized graph constraint for reachability or for acyclicity, and as these constraints dominate the encoding, they can have a strong impact on the scalability of the approach [PR18].

19.7.3. Planning with Full Observability in NP

Earlier works had suggested that contingent planning in general either requires a formalism that is harder than NP, or must be dramatically limited in terms of the size of the plans, the length of the executions, and the compactness of the transition system representation.

However, Geffner and Geffner [GG18] demonstrated that for a class of contingent planning problems with full observability, plans and their executions can be represented as a propositional formula even with the actions and states represented compactly in terms of state variables.

In this formula, each node of a plan is associated with atomic propositions that describe the relevant features of the state when the execution is in that node. This allows each satisfying assignment of the formula to represent both a branching plan and all of its relevant executions, thereby bringing the complexity down to NP and enabling reductions to SAT.

This approach resembles the preceding one (Section 19.7.2) in identifying executions with a graph that is represented in the satisfying assignments of the formula, and by relying on graph constraints to express the conditions for the validity of a plan.

The approach seems to be limited by the requirement that all states in a given node of the plan have to agree on the values of some of the state variables, but many practical problems are still effectively representable in this way.

References

- [ABC⁺02] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT based approach for solving formulas over Boolean and linear mathematical propositions. In A. Voronkov, editor, *Automated Deduction - CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark, July 27-30, 2002, Proceedings*, number 2392 in Lecture Notes in Computer Science, pages 195–210. Springer-Verlag, 2002.
- [ABH⁺97] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Partial-order reduction in symbolic state space exploration. In *Computer Aided Verification, 9th International Conference, CAV'97, Haifa, Israel, June 22-25, 1997, Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 340–351. Springer-Verlag, 1997.
- [BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In W. R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems, Proceedings of 5th International Conference, TACAS'99*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 1999.

- [BF97] A. L. Blum and M. L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.
- [BR05] M. Büttner and J. Rintanen. Satisfiability planning with constraints on the number of actions. In S. Biundo, K. Myers, and K. Rajan, editors, *ICAPS 2005. Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling*, pages 292–299. AAAI Press, 2005.
- [Bra01] R. I. Brafman. On reachability, relevance, and resolution in the planning as satisfiability approach. *Journal of Artificial Intelligence Research*, 14:1–28, 2001.
- [Byl94] T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.
- [CCD16] K. Chatterjee, M. Chmelik, and J. Davies. A symbolic SAT-based algorithm for almost-sure reachability with small strategies in POMDPs. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI-16)*, pages 3225–3232. AAAI Press, 2016.
- [DK01] M. B. Do and S. Kambhampati. Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP. *Artificial Intelligence*, 132(2):151–182, 2001.
- [DNK97] Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding planning problems in nonmonotonic logic programs. In S. Steel and R. Alami, editors, *Recent Advances in AI Planning. Fourth European Conference on Planning (ECP'97)*, number 1348 in Lecture Notes in Computer Science, pages 169–181. Springer-Verlag, 1997.
- [EMW97] M. Ernst, T. Millstein, and D. S. Weld. Automatic SAT-compilation of planning problems. In M. Pollack, editor, *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 1169–1176. Morgan Kaufmann Publishers, 1997.
- [GG18] T. Geffner and H. Geffner. Compact policies for non-deterministic fully observable planning as sat. In *ICAPS 2018. Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling*, pages 88–96. AAAI Press, 2018.
- [God91] P. Godefroid. Using partial orders to improve automatic verification methods. In E. M. Clarke, editor, *Proceedings of the 2nd International Conference on Computer-Aided Verification (CAV '90), Rutgers, New Jersey, 1990*, number 531 in Lecture Notes in Computer Science, pages 176–185. Springer-Verlag, 1991.
- [GS98] A. Gerevini and L. Schubert. Inferring state constraints for domain-independent planning. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, pages 905–912. AAAI Press, 1998.
- [HJ00] P. Haslum and P. Jonsson. Some results on the complexity of planning with incomplete information. In S. Biundo and M. Fox, editors, *Recent Advances in AI Planning. Fifth European Conference on Planning (ECP'99)*, number 1809 in Lecture Notes in Artificial Intelligence, pages 308–318. Springer-Verlag, 2000.

- [KMS96] H. Kautz, D. McAllester, and B. Selman. Encoding plans in propositional logic. In L. C. Aiello, J. Doyle, and S. Shapiro, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference (KR '96)*, pages 374–385. Morgan Kaufmann Publishers, 1996.
- [KS92] H. Kautz and B. Selman. Planning as satisfiability. In B. Neumann, editor, *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 359–363. John Wiley & Sons, 1992.
- [KS96] H. Kautz and B. Selman. Pushing the envelope: planning, propositional logic, and stochastic search. In *Proceedings of the 13th National Conference on Artificial Intelligence and the 8th Innovative Applications of Artificial Intelligence Conference*, pages 1194–1201. AAAI Press, August 1996.
- [KW99] H. Kautz and J. Walser. State-space planning by integer optimization. In *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI-99) and the 11th Conference on Innovative Applications of Artificial Intelligence (IAAI-99)*, pages 526–533. AAAI Press, 1999.
- [Lit97] M. L. Littman. Probabilistic propositional planning: Representations and complexity. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97) and 9th Innovative Applications of Artificial Intelligence Conference (IAAI-97)*, pages 748–754. AAAI Press, 1997.
- [ML03] S. M. Majercik and M. L. Littman. Contingent planning under uncertainty via stochastic satisfiability. *Artificial Intelligence*, 147(1-2):119–162, 2003.
- [MR91] D. A. McAllester and D. Rosenblitt. Systematic nonlinear planning. In T. L. Dean and K. McKeown, editors, *Proceedings of the 9th National Conference on Artificial Intelligence*, pages 634–639. AAAI Press / The MIT Press, 1991.
- [PB91] C. Pixley and G. Beihl. Calculating resettability and reset sequences. In *Computer-Aided Design, 1991. ICCAD-91. Digest of Technical Papers., 1991 IEEE International Conference*, pages 376–379, 1991.
- [Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE, 1977.
- [PR18] B. Pandey and J. Rintanen. Planning for partial observability by SAT and graph constraints. In *ICAPS 2018. Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling*, pages 190–198. AAAI Press, 2018.
- [RGPS09] N. Robinson, C. Gretton, D.-N. Pham, and A. Sattar. SAT-based parallel planning using a split representation of actions. In *ICAPS 2009. Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*, pages 281–288. AAAI Press, 2009.
- [RGS15] M. F. Rankooh and G. Ghassem-Sani. ITSAT: an efficient SAT-based temporal planner. *Journal of Artificial Intelligence Research*, 53:541–632, 2015.
- [RHN06] J. Rintanen, K. Heljanko, and I. Niemelä. Planning as satisfiability:

- parallel plans and algorithms for plan search. *Artificial Intelligence*, 170(12-13):1031–1080, 2006.
- [Rin98] J. Rintanen. A planning algorithm not based on directional search. In A. G. Cohn, L. K. Schubert, and S. C. Shapiro, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR '98)*, pages 617–624. Morgan Kaufmann Publishers, June 1998.
- [Rin99] J. Rintanen. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.
- [Rin04] J. Rintanen. Complexity of planning with partial observability. In *ICAPS 2004. Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling*, pages 345–354. AAAI Press, 2004.
- [Rin07] J. Rintanen. Asymptotically optimal encodings of conformant planning in QBF. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI-07)*, pages 1045–1050. AAAI Press, 2007.
- [Rin08] J. Rintanen. Regression for classical and nondeterministic planning. In M. Ghallab, C. D. Spyropoulos, and N. Fakotakis, editors, *ECAI 2008. Proceedings of the 18th European Conference on Artificial Intelligence*, pages 568–571. IOS Press, 2008.
- [Rin12a] J. Rintanen. Engineering efficient planners with SAT. In *ECAI 2012. Proceedings of the 20th European Conference on Artificial Intelligence*, pages 684–689. IOS Press, 2012.
- [Rin12b] J. Rintanen. Planning as satisfiability: heuristics. *Artificial Intelligence*, 193:45–86, 2012.
- [Rin17] J. Rintanen. Temporal planning with clock-based SMT encodings. In *IJCAI 2017, Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 743–749. AAAI Press, 2017.
- [Sac75] E. D. Sacerdoti. The nonlinear nature of plans. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence*, pages 206–214, 1975.
- [SD05] J.-A. Shin and E. Davis. Processes and continuous change in a SAT-based planner. *Artificial Intelligence*, 166(1):194–253, 2005.
- [SM73] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time. In *Proceedings of the 5th Annual ACM Symposium on Theory of Computing*, pages 1–9, 1973.
- [Tur02] H. Turner. Polynomial-length planning spans the polynomial hierarchy. In *Logics in Artificial Intelligence, European Conference, JELIA 2002*, number 2424 in Lecture Notes in Computer Science, pages 111–124. Springer-Verlag, 2002.
- [Val91] A. Valmari. Stubborn sets for reduced state space generation. In G. Rozenberg, editor, *Advances in Petri Nets 1990. 10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany*, number 483 in Lecture Notes in Computer Science, pages 491–515. Springer-Verlag, 1991.
- [vBC99] P. van Beek and X. Chen. CPlan: A constraint programming approach to planning. In *Proceedings of the 16th National Conference on Arti-*

ficial Intelligence (AAAI-99) and the 11th Conference on Innovative Applications of Artificial Intelligence (IAAI-99), pages 585–590. AAAI Press, 1999.

- [VBLN99] T. Vossen, M. Ball, A. Lotem, and D. Nau. On the use of integer programming models in AI planning. In T. Dean, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 304–309. Morgan Kaufmann Publishers, 1999.
- [WW99] S. A. Wolfman and D. S. Weld. The LPSAT engine & its application to resource planning. In T. Dean, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 310–315. Morgan Kaufmann Publishers, 1999.

This page intentionally left blank

Chapter 20

Software Verification

Daniel Kroening

20.1. Programs use Bit-Vectors

Ensuring correctness or quality of computer software is an increasingly difficult task due to the ever-growing size and complexity of programs. Correctness is especially important in the case of software that runs on computers embedded in our transportation and communication infrastructure. Examples of software errors that led to huge financial losses and loss of life are easy to find.

From the point of view of a computer scientist, these quality problems are not easy to explain, as reasoning about sequential programs is possibly the oldest discipline in computer science. Programs are well-understood as a set of instructions that manipulate the values of a given set of program variables. When reasoning about a program, e.g., within a program analyzer, these program variables are typically modeled as unbounded integers.

In reality, program analyzers rarely need to reason about unbounded integers. With a few exceptions, most programming languages offer basic data types that have *bit-vector semantics*, i.e., they have a bounded range defined by the number of bits allocated for them. In case of an unsigned number with n bits, this range is $0, \dots, 2^n - 1$. If this range is exceeded by an arithmetic operation, the result *wraps around*, which means that the modulo 2^n of the actual result is computed.

But it is not modular arithmetic that is needed: All commercially available processors implement bit-wise operators, such as shifting or a bit-wise logical ‘AND’. Such operators are inexpensive to implement in hardware, and the corresponding instructions are executed more efficiently than arithmetic instructions. As a consequence, many programs use such instructions, e.g., for performance-critical operations such as encodings of data streams or the like. Some bit-wise operators have reasonable arithmetic models when interpreted as operators over integers, e.g., a one-bit left-shift corresponds to a multiplication by two. However, most bit-wise operators require modeling with non-linear integer arithmetic if interpreted as functions over integers. Due to these issues, many program analysis engines are unsound with respect to the actual semantics of arithmetic over bit-vectors.

A possible solution is to model programs by closely following the hardware that is used to implement their execution environment: bit-vectors can be modeled as a set of Boolean variables, and bit-vector operators as a set of Boolean functions. Propositional SAT is therefore an accurate and efficient decision engine for decision problems that arise in program verification.

In this chapter, we describe methods for program verification using a SAT solver as decision procedure. For a broader perspective on program verification, see [DKW08]. In Section 20.2, we first describe how to automatically extract formal models of software from program source code. We use ANSI-C as programming language for the examples, but all techniques that we describe are also applicable to languages that are similar, e.g., JAVA or C++. In Section 20.3, we explain how to transform typical program expressions into propositional logic, making them suitable as input for a SAT solver. In Section 20.4, we then extend Bounded Model Checking (BMC), a hardware verification technique, to apply to software. BMC is often limited to refutation. *Predicate Abstraction* is a technique that is geared at proving control-flow dominated properties of programs. We describe a SAT-based variant of predicate abstraction in Section 20.5.

20.2. Formal Models of Software

The program that is to be analyzed is typically modeled by means of an state transition system.

Definition 1 (Transition System) *A Transition System is a triple (S, S_0, R) , where S is a set of states, $S_0 \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is the transition relation.*

The transition system (the model) of the program is denoted by M . A program state $s \in S$ comprises of an encoding of the current program location and a valuation of the (global and local) program variables. We denote the finite set of program locations by \mathcal{L} , and the program location of state $s \in S$ by $s.l$. A particular program location, denoted by ℓ_0 , is designated as the entry point (main function).

We assume that there is a set of variables V over a finite domain D . The set V is possibly unbounded if dynamic allocation of objects is permitted. We use $s.v$ to denote the value of a variable $v \in V$ in state s .

Function calls are typically modeled by means of an unbounded function call stack $\Gamma : \mathbb{N} \rightarrow D \dot{\cup} \mathcal{L}$, which allows recursion. The stack can be used to save call locations (i.e., elements from \mathcal{L}) and local variables in order to preserve locality (i.e., elements from D). We write $s.\Gamma$ to denote the stack in state s . In summary, S consists of three components:

$$S = \mathcal{L} \times \underbrace{(V \rightarrow D)}_{\text{Variables}} \times \underbrace{(\mathbb{N} \rightarrow (D \dot{\cup} \mathcal{L}))}_{\text{Stack}}$$

This definition only permits one program counter, and thus, only one thread. An initial state in S_0 assigns the initial program location ℓ_0 to the program

counter. The initialization of global data is assumed to be performed as part of the program, and is not encoded in S_0 .

The instructions of the program are modeled by means of the transition relation R : given a pair of states $\langle s, s' \rangle \in S \times S$, $R(s, s')$ holds if and only if there is a transition from s to s' . This encodes both the control flow (by means of constraints on the program counters in s and s'), and the changes in data and of the call stack. Following the structure of the program, R is partitioned into separate relations R_l , one for each program location $l \in \mathcal{L}$. The appropriate R_l is chosen by means of a case-split on $s.l$, i.e.:

$$R(s, s') : \iff \bigwedge_{l \in \mathcal{L}} (s.l = l \longrightarrow R_l(s, s')) \quad (20.1)$$

The transformation of a program given in a typical sequential programming language into this form is straight-forward.

```
extern int  getch();
extern int  printf(const char *, ... );

int main( void ) {
    char x;
    x = getch();
    while (x != '\n') {
        switch(x) {
            case 'a':
            case 'b':
                printf("a_or_b");
                break;
            case 'c':
                printf("c_and_");
                /* fall-through */
            default:
                printf("d");
                break;
        }
    }
    return 0;
}
```

Figure 20.1. A fragment of a C program

Example 1 *As an example, consider the ANSI-C code fragment given in Fig. 20.1. The control flow graph is shown in Fig. 20.2. The variable `getch$1` is an auxiliary variable added as part of the translation process. It contains the return value of the `getch()` function. Similar variables are used to eliminate any side-effects from the expressions, e.g., the compound assignment operators.*

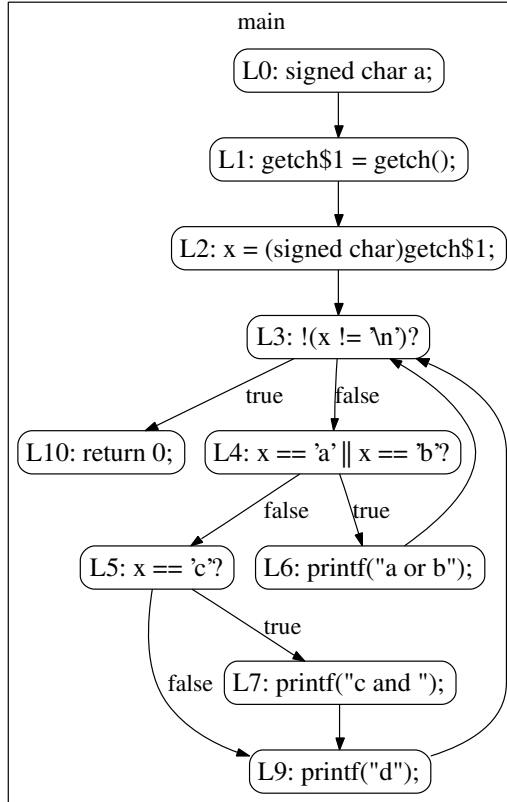


Figure 20.2. Control flow graph of the C program in Fig. 20.1: the nodes in the graph correspond to the program locations.

Each node in the graph corresponds to one program location in \mathcal{L} . Consider the assignment to x in location $L2$. The corresponding transition relation has four conjuncts:

$$\begin{aligned}
 R_{L2}(s, s') &\iff s'.l = L3 \\
 &\quad \wedge \forall v \neq x : s'.v = s.v \\
 &\quad \wedge s'.x = TC(s.\text{getch}\$1) \\
 &\quad \wedge s'.\Gamma = s.\Gamma
 \end{aligned}$$

The first conjunct corresponds to the unconditional edge from node $L2$ to $L3$. The second conjunct enforces that the values of all variables but x do not change in the transition from s to s' . The third conjunct defines the new value of x . The function TC is a mapping from the signed integers to the type `char`, which corresponds to a modulo 256 operation on most architectures. Finally, the fourth conjunct states that the stack does not change.

As an example of a branch, consider program location $L3$. If x is equal to `'\n'`, the **while** loop terminates and program proceeds to location $L10$. Otherwise, the

program continues with location $L4$. Formally, this is captured by a case-split on the value of x in state s :

$$R_{L3}(s, s') \iff s'.\ell = \begin{cases} L10 : s.x = \wedge n' \\ L4 : \textit{otherwise} \end{cases} \\ \wedge \forall v \in V. s'.v = s.v \\ \wedge s'.\Gamma = s.\Gamma$$

We write $s \longrightarrow t$ as a shorthand for $R(s, t)$, and we write $s \rightsquigarrow t$ as shorthand for $R^*(s, t)$, where R^* denotes the reflexive transitive closure of R . The main property of interest in software verification is reachability of certain 'bad' program locations, called *error locations*.¹ This is formalized by the following definition.

Definition 2 (Counterexample) We denote the set of error locations by $\mathcal{L}_E \subset \mathcal{L}$. An error state is a state $s \in S$ with $s.\ell \in \mathcal{L}_E$. A counterexample for a reachability property is a sequence of states s_0, s_1, \dots, s_n with $s_i \in S$, $s_0 \in S_0$, $s_n.\ell \in \mathcal{L}_E$, and $s_j \longrightarrow s_{j+1}$ for $0 \leq j < n$.

20.3. Turning Bit-Vector Arithmetic into CNF

Program analysis algorithms generate verification conditions, which are formulas in the combined expression and assertion logic of the programming language that is used. Let ψ denote such a verification condition.

20.3.1. Pointers

In addition to the usual arithmetic and bit-level operators, we also consider verification conditions ψ that contain operators related to pointers and (possibly unbounded) arrays. The pointer and array operators are usually removed as a pre-processing step by means of a reduction to bit-vector arithmetic. Note that reference types (as offered by JAVA or C++) are simply pointer types with different syntax and a restricted set of operators.

The transformation of pointer-typed variables is commonly done by replacing each pointer-typed variable in ψ by a pair $\langle o, i \rangle$ of bit-vectors. The first component identifies the object that the pointer points to. The second component denotes the offset within the object that the pointer points to. Note that an array is considered to be only one object, independently of how many elements it may have. Therefore, ψ can only contain a finite number of objects, and a finite-width bit-vector is sufficient to enumerate them.

The pointer dereferencing operators require some form of alias analysis. Alias analysis is performed in every modern optimizing compiler, and the algorithms implementing it are well-optimized. Such an analysis can be applied in the context of program verification as well: the output of the alias-analysis procedure is a function that maps the pointers in the program to the set of objects that the pointer may point to. We denote this map by \mathcal{V} . Using this map, the dereferencing

¹Many other properties, e.g., assertions, array-bounds, memory-safety, can be trivially rewritten as reachability problems.

operators in the formula can be replaced by a simple case-split. Let $\mathcal{V}(p)$ be $\{x_1, x_2, \dots, x_n\}$. We substitute each occurrence of $*p$ in ψ as follows:

$$*p = \begin{cases} x_1 : p.o = (\&x_1).o \\ x_2 : p.o = (\&x_2).o \\ \dots \\ x_n : \text{otherwise} \end{cases}$$

Recall that $p.o$ denotes the object that a pointer p points to. In case of an x_i of array type, the index into the array is $p.i$. The other pointer-related operators are transformed as follows:

- The address-of operator $\&x$ returns a constant that uniquely identifies the object x . The numbering can be done in an arbitrary manner, as languages such as ANSI-C do not guarantee an ordering of the objects in memory. Expressions such as $\&x[i]$ can be transformed into $\&x[0] + i$.
- The NULL pointer is a constant with a unique object number.
- The formula ψ may contain calls to procedures that allocate memory dynamically, e.g., `malloc()` in case of ANSI-C, or `new` in case of the object-oriented languages. A new variable of the appropriate type is generated for each one of these calls. The call to `malloc()` or `new` is subsequently replaced by a (constant) pointer pointing to the corresponding new variable.
- Pointer arithmetic $p + z$ and $p - z$ is simply arithmetic on the offset component of p . ANSI-C does not permit pointer arithmetic to exceed the bounds of an object.
- Equality on pointers is equality of the corresponding pairs.
- The relational operators on pointers $p < q$ are replaced by the corresponding relations on the offsets. ANSI-C does not permit relations between pointers that point to different objects.
- The difference $p - q$ of two pointers is the difference of the offsets of p and q . Again, ANSI-C does not permit computing the difference of two pointers that point to different objects.
- Casts between pointer types do not change the value of the pointer.

The alias analysis required for the pointer dereferencing operators has to be performed prior to deciding validity of the verification condition, and depends on the actual program analysis engine, two of which are described later.

20.3.2. Arrays

If of finite size, array-typed variables and the corresponding array-operators can be turned trivially into very large bit-vectors. However, this transformation is usually only efficient if the size of the array is rather small, and infeasible if the size of the array is not a constant. In case of large arrays, or arrays that have variable size, array expressions are typically removed using a reduction to *uninterpreted functions*, which in turn, can be reduced to bit-vector logic by adding equality expressions. The details of this step are described in Part 2, Chapter 33. The resulting formula only contains bit-vector-typed variables and operators over bit-vectors.

20.3.3. Deciding Bit-Vector Arithmetic

There is a large body of work on efficient solvers for bit-vector arithmetic. The earliest work is based on algorithms from the theorem proving community, and uses a canonizer and solver for the theory. The work by Cyrluk et al. [CMR97] and by Barrett et al. on the Stanford Validity Checker [BDL98] fall into this category. These approaches are very elegant, but are restricted to a subset of bit-vector arithmetic comprising concatenation, extraction, and linear equations (not inequalities) over bit-vectors. Non-linear operations, including unrestricted bit-wise operations, are typically not supported.

Bit-Blasting With the advent of efficient propositional SAT solvers such as ZChaff [MMZ⁺01], these approaches have been obsoleted. The most commonly applied approach to check satisfiability of these formulas is to replace the arithmetic operators by circuit equivalents to obtain a propositional formula. This propositional formula is then converted into CNF using a Tseitin encoding (see Part 1, Chapter 2: CNF encodings). The CNF formula is then given to the propositional SAT solver. This approach is commonly referred to as 'bit-blasting', as the word-level structure is lost. It is used by many software analysis tools, e.g., by Alloy [Jac06]. We now provide a short tutorial on this technique.

We denote the bit-vector formula we want to convert by ψ , and the result of the bit-blasting procedure (the propositional formula) by ψ_f . Let $A(\psi)$ denote the set of atoms in ψ . As a first step, each atom $a \in A(\psi)$ is replaced by a new Boolean variable, denoted by $\mu(a)$. The resulting formula is denoted by ψ_{sk} , and is called the *propositional skeleton* of ψ . The propositional skeleton is the expression that is assigned to ψ_f initially.

The algorithm then assigns a vector of new Boolean variables to each bit-vector term in ψ . We use $\mu(t)$ to denote this vector of variables for a given term t , and $\mu(t)_i$ to denote the variable for the bit with index i of the term t . The algorithm then computes a constraint for each unique term in t that is not a variable. The constraint depends on the operator in the term.

The constraint for bit-wise operators is straight-forward. As an example, consider bit-wise OR, i.e., t is $a \mid b$, and let both a and b have l bits. The constraint added to ψ_f is:

$$\bigwedge_{i=0}^{l-1} ((a_i \vee b_i) \iff \mu(t)_i) \quad (20.2)$$

The constraints for other bit-wise operators follow the same pattern. The constraint for the arithmetic operators (addition, subtraction, and so on) follows simplistic implementations of these operators as a circuit.

The Cogent procedure is an implementation of bit-blasting that takes ANSIC expressions as input [CKS05a]. Most other decision procedures use simple variants of this technique. CVC-Lite pre-processes the input formula using a normalization step followed by equality rewrites before finally bit-blasting to SAT [BGD05]. Wedler et al. have a similar approach wherein they normalize bit-vector formulas in order to simplify the generated SAT instance [WSK05]. STP [CGP⁺06], which is the engine behind the EXE program analyzer, is a successor to the CVC-Lite system; it performs several array optimizations, as well

as arithmetic and Boolean simplifications on the bit-vector formula before bit-blasting. Yices [DdM06] applies bit-blasting to all bit-vector operators except for equality. Bryant et al. present a method to solve hard instances that is based on iterative abstraction refinement [BKO⁺07].

20.4. Bounded Model Checking for Software

20.4.1. Background on BMC

Bounded model checking (BMC) is one of the most commonly applied formal verification techniques for circuitry in the semiconductor industry. The technique owes this success to the impressive capacity of propositional SAT solvers. The design under verification is unwound k times together with a correctness property to form a propositional formula, which is then passed to the SAT solver. The formula is satisfiable if and only if there is a trace of length k that refutes the property. The technique is inconclusive if the formula is unsatisfiable, as there might still be a counterexample that is longer than k steps.

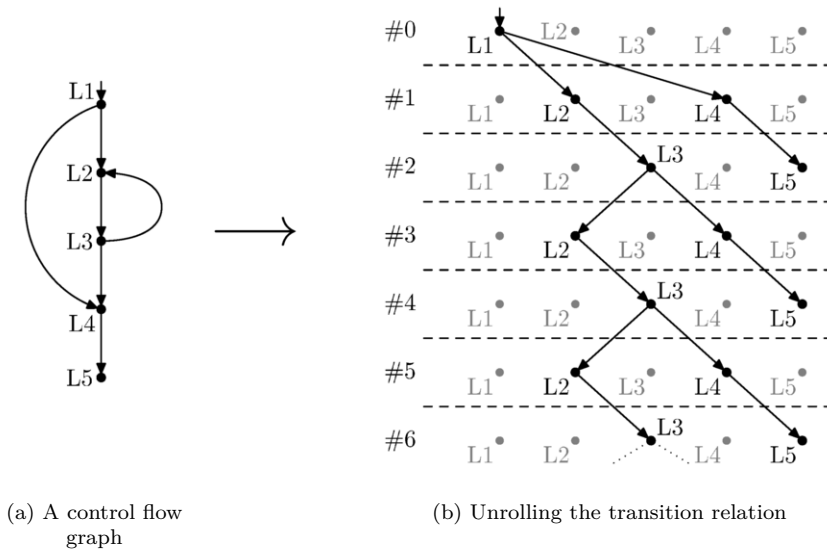
BMC encodings have been proposed for a wide range of specification logics, including LTL and CTL. The presentation that follows is restricted to reachability properties. As defined in Sec. 20.2, we use R to denote the relation that describes the set of possible steps (transitions) the design can make. The set of initial states of the system is given by S_0 , and we use the predicate p as a short-hand for the reachability property of interest. To obtain a BMC instance with k steps, the transition relation is replicated k times. The variables in each replica are renamed such that the next state of step k is used as current state of step $k + 1$. The transition relations are conjoined. In addition to that, the current state of the first step is constrained to be in S_0 , and one of the states must satisfy $\neg p$:

$$\begin{array}{ccccccc} S_0 \wedge R & \xrightarrow{\wedge} & R & \xrightarrow{\wedge} & \dots & \xrightarrow{\wedge} & R \\ \bullet & \xrightarrow{\quad} & \bullet & \xrightarrow{\quad} & \bullet & \xrightarrow{\quad} & \bullet \\ \neg p & \vee & \neg p & \vee & \neg p & \vee & \neg p \end{array}$$

Any satisfying assignment to such a formula corresponds to a path beginning in the initial state that ends in a state that violates p , and thus, is a counterexample to p . The size of this formula is linear in the size of the design and in k .

20.4.2. Unwinding the Entire Program at once

The idea of BMC is also applicable to system-level software. The most straightforward manner to implement BMC for software is to treat the entire program as one transition relation R , and thus, to ignore the partitioning given by the program location. The program counter ℓ can be represented using a bit-vector of width $\lceil \log_2 |\mathcal{L}| \rceil$ using a binary encoding, or with a one-hot encoding, which uses one bit for each program location. An additional optimization is to merge instructions that form a basic block into a single program location, which reduces the number of locations.



(a) A control flow graph

(b) Unrolling the transition relation

Figure 20.3. When unrolling the entire transition relation (a), the unreachable nodes (in grey) may be omitted (b).

We denote the state in time-frame i of the unwinding by s_i , and construct the following formula: state s_0 must be an initial state, there must be a transition between s_i and s_{i+1} for any i , and one of the s_i must reach an error state.

$$S_0(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigvee_{i=0}^k \bigvee_{l \in \mathcal{L}_E} s_i.l = l \quad (20.3)$$

This formula can then be passed to a decision procedure, e.g., the procedure for deciding bit-vector logic described in Sec. 20.3. If it is satisfiable, the solver is also expected to provide a satisfying assignment. From this satisfying assignment, we can extract the values of s_0, \dots, s_k , which is a counterexample according to Def. 2. If Eq. 20.3 is unsatisfiable, we can conclude that no error state is reachable in k or less steps.

Optimizations The size of Eq. 20.3 is k times the size of R , i.e., the size of the program. For large programs, this is already prohibitive, and thus, several optimizations have been proposed. The first step is to perform an analysis of the control flow that is possible within the bound. Consider the small control flow graph in Fig. 20.3a. Each node corresponds to one basic block; the edges correspond to possible control flow between the blocks. Note that block L1 can only be executed in the very first step of any path. Similarly, block L2 can only be executed in step 2, 4, 6 and so on. This is illustrated in Fig. 20.3b: the nodes unreachable in the corresponding time-frames are in grey. This optimization can be seen as the equivalent of the bounded cone of influence (B-COI) reduction [BCRZ99].

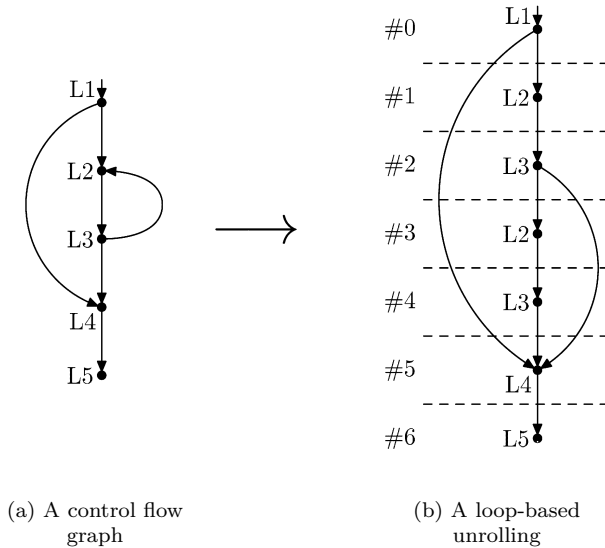


Figure 20.4. A loop-based unrolling of the control flow graph in Fig. 20.3a

20.4.3. Unwinding Loops Separately

Again, consider the example in Fig. 20.3a: observe that any path through the CFG contains L4 and L5 at most once. However, note that the unwinding of the transition relation in Fig. 20.3b contains a total of three copies of L4 and L5. This motivates the idea of *loop unwinding*. Instead of unwinding the entire transition relation, each loop is unwound separately. Loop unwinding is illustrated in Fig. 20.4: the loop between L2 and L3 is unwound twice. The algorithm maintains a separate unrolling bound for each loop of the program. In case of nested loops, a separate bound can be used even for each *instance* of a specific loop.

As a consequence, the transition relation R can no longer simply be replicated k times. Instead, the control-flow of the original program is modified by unwinding the loops. Unwinding a loop k times corresponds to replicating the body of the loop k times. Each copy is guarded by an appropriate `if` statement in order to handle loops whose execution aborts before k iterations.

The example in Fig. 20.5 shows an unwinding with $k = 2$. This permits loops where the number of actual iterations depends on a non-deterministically chosen input. The last `if` statement is augmented with an `else` branch. The `else` branch corresponds to the case that k is too low for this loop, and not enough unwinding was performed. The `assume(false)` statement ensures that this does not result in spurious paths. Appropriate care must be taken to preserve the locality of variables that are declared within the loop body. Recursive function calls, and loops built by means of backward `goto` statements can be unwound in the same manner as explicit loops.

$$\begin{array}{l}
 \text{while}(x) \\
 \text{BODY;}
 \end{array}
 \longrightarrow
 \begin{array}{l}
 \text{if}(x) \{ \\
 \text{BODY;} \\
 \text{while}(x) \\
 \text{BODY;} \\
 \}
 \end{array}
 \longrightarrow
 \begin{array}{l}
 \text{if}(x) \{ \\
 \text{BODY;} \\
 \text{if}(x) \\
 \text{BODY;} \\
 \text{else} \\
 \text{assume(false);} \\
 \}
 \end{array}$$

Figure 20.5. A loop-based unrolling of a **while** loop with depth two.

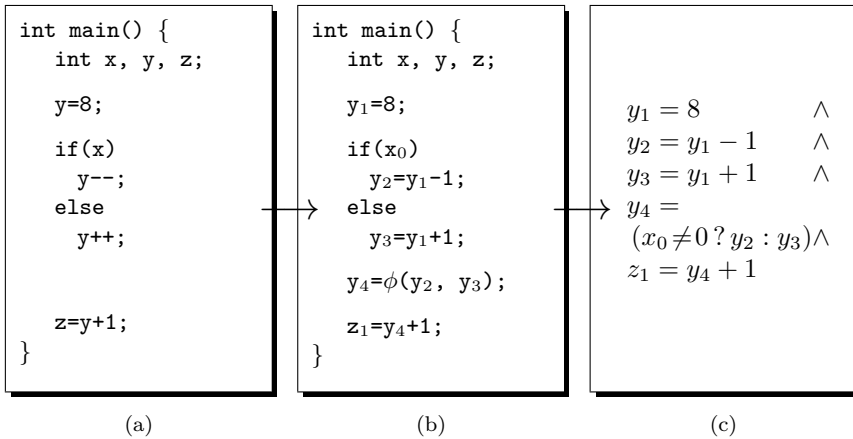


Figure 20.6. The program (a) is transformed into SSA (b). The merge-point in the control flow graph following the **if** statement is turned into a ϕ -node. The SSA is transformed into a formula (c).

The next step after unwinding is to transform the program into static single assignment (SSA) form [AWZ88]. This operation is also found in most modern optimizing compilers, and thus, it is easy to find efficient implementations. We illustrate this transformation by example: Fig. 20.6 shows a small C program and its SSA form. The SSA form is built by numbering the variables according to the number of times they have been assigned so far. At merge points within the control flow graph, the appropriate value is chosen using a ϕ -function, which is simply a case-split using the branch condition. It is straight-forward to turn the SSA-form into a formula: one simply reads the assignments as equality, and conjoins them.

There are alternatives for generating formulas from programs, e.g., a path-based exploration, which amounts to building a separate formula for each program path. Loop unwinding, as described above, is not the same as a path-based exploration: in the example, the path that corresponds to the branch from L1 to L4 merges with the path that follows the loop. As a consequence, the formula that is generated is *linear* in the number of loop unrollings and in the size of the

program, even if there is an exponential number of paths through the CFG.

Compared to unrolling the entire program, the loop-based unrolling may result in formulas that are more compact. The loop-based unrolling also requires fewer case-splits in the formula, as the program location is fixed for a particular part of the formula. A disadvantage of the loop-based unwinding is that it may require more time-frames in order to reach a given control flow location. In the example in Fig. 20.4, the unwinding of the transition relation reaches iteration three in the loop between L2 and L3, whereas the loop-based unrolling only reaches iteration two with the same number of steps.

20.4.4. A Complete BMC for Software

Bounded Model Checking, when applied as described above, is inherently incomplete, as it searches for violations of the property only up to the given bound. Bugs that are deeper than the given bound are missed. Nevertheless, BMC can be used to *prove* safety properties if applied in a slightly different way.

Intuitively, if we could perform the search *deep enough*, we can guarantee that we have already examined all the relevant behavior of the model, and that searching any deeper only exhibits states that we have explored already. A depth that provides such a guarantee is called a *completeness threshold* [KS03]. Computing the smallest such threshold is as hard as Model Checking itself, and thus, one settles for over-approximations in practice.

In the context of software, one way to obtain a depth-bound for the program is to determine a high-level worst-case execution time (WCET). This time is given in terms of a bound on the maximum number of loop-iterations. Tools that compute a WCET often perform a simplistic syntactic analysis of loop structures. If that fails, an iterative algorithm can be applied: One first guesses a bound, and then checks an assertion that is violated by paths that exceed this bound [KCY03, CKL04]. If so, the bound is increased, and the procedure starts over. Such an assertion is called an *unwinding assertion*, and is obtained by simply replacing the `assume(false)` statement as explained above by an `assert(false)` statement. This technique is applicable if the program (or at least the body of its main loop) has a runtime-bound, which is highly desirable for many embedded applications.

20.4.5. Tools that Implement BMC for Software

There are a number of implementations of BMC available for software verification. The first implementation of a depth-bounded symbolic search in software was reported in 2000 by Currie et al. [CHR00].

The CBMC tool, developed at CMU, emulates a wide range of architectures as environment for the program under test [KCY03, CKL04]. It supports both little-endian and big-endian memory models, and the header files needed for Linux, Windows, and Mac-OS X. It implements loop unrolling as described in Sec. 20.4.3. It uses bit-flattening, or bit-blasting, to decide the resulting bit-vector formula. It also has options to export the formula in a variety of word-level formats. It is the only tool that also supports C++, SpecC and SystemC. The main application of CBMC is checking consistency of system-level circuit models given in C or

SystemC with an implementation given in Verilog. IBM has developed a version of CBMC for concurrent programs [RG05].

The only tool that implements an unwinding of the entire transition system, as described in Sec. 20.4.2, is F-SOFT [ISGG05], developed at NEC Research. It also features a SAT solver that is customized to the decision problems that are generated by BMC. The benchmarks that have been reported are system-level UNIX applications such as `pppd`.

A number of variants of these implementations have been reported. Armando et al. implement a version of CBMC that generates a decision problem for an SMT solver for integer linear arithmetic [AMP06].

SATURN is a very specialized implementation of BMC: it is custom-tailored to the properties it checks [XA05]. It implements loop unwinding as described in Sec. 20.4.3. The authors have applied it to check two different properties of Linux kernel code: NULL-pointer dereferences and locking API conventions. They demonstrate that the technique is scalable enough to analyze the entire Linux kernel. Soundness is relinquished for performance; SATURN performs at most two unwindings of each loop. Bugs that are only revealed with more than two loop iterations are missed. This approach is implemented by numerous software analysis tools as well, e.g., by ESC/Java [FLL⁺02].

The EXE tool is also specialized to bug-hunting [YST⁺06]. It combines explicit execution and path-wise symbolic simulation in order to detect bugs in system-level software such as file system code. It uses a very low-level memory model, which permits checking programs that rely on specific memory layouts.

While BMC is probably the best technique for bug-hunting, other techniques are better at *proving* properties. Predicate abstraction excels at the verification of light-weight control-flow dominated properties, and is discussed next.

20.5. Predicate Abstraction using SAT

20.5.1. The Need for Abstraction

In most cases, proving correctness of the entire code is not the goal of automatic software verification. To begin with, writing a sufficiently strong specification of all program behavior is already very hard. Many properties of programs that are of interest for automatic verification are therefore very light-weight.

As an example of a light-weight property, consider the pre-conditions of a locking-API, such as offered by the PTHREAD library. The `pthread_mutex_lock(m)` function attempts to acquire a given mutex m , while the `pthread_mutex_unlock(m)` releases the lock m . Similar functions are provided by many other APIs. A program using this API must obey certain restrictions on its use:

- A process holding a lock must not attempt to acquire it again before releasing it,
- a process must not release a lock unless it holds it.

As a consequence of the two rules above, a program using the API is expected to call the lock/unlock functions in an alternating fashion. E.g., two consecutive calls to `pthread_mutex_lock(m)` with the same mutex m are erroneous.

While bugs related to concurrency can have serious consequences, it is obvious that locking and unlocking mutexes is not the main purpose of the program. The program likely contains many more instructions, which are irrelevant with respect to the lock/unlock property. The code that can influence the calls to the lock/unlock functions likely comprises only a small fraction of the total amount of code. In case of the lock/unlock property, the right order of calls is often already apparent from the control flow graph, completely disregarding any data.

The goal of *abstraction* is therefore to construct a model that only contains those parts of the program that are actually relevant to the property of interest. As this model is often much smaller than the original program, the verification becomes more efficient. The trick is to be sure that the reduction is done in a *conservative manner*, i.e., that details relevant to the property are preserved. We describe a systematic approach to construct such abstractions next.

20.5.2. Existential Abstraction

Formally, the goal of abstraction is to compute an abstract model \hat{M} from the concrete model M such that the size of the state-space is reduced while the property of interest is preserved. We denote the set of abstract states by \hat{S} . A concrete state is mapped to an abstract state by means of an *abstraction function*, which we denote by $\alpha : S \rightarrow \hat{S}$. We also extend the definition of α to sets of states: Given a set $S' \subseteq S$, we denote $\{\hat{s} \in \hat{S} \mid \exists s \in S'. \alpha(s) = \hat{s}\}$ by $\alpha(S')$. The inverse of α , called the *concretization function*, maps abstract states back to the corresponding concrete states, and is denoted by γ [CC77]:

$$\gamma(\hat{S}') := \{s \in S \mid \alpha(s) \in \hat{S}'\} \quad (20.4)$$

As mentioned above, we restrict the presentation to *safety properties*. The goal therefore is to compute an abstraction that preserves safety properties: any program location that is reachable in M must be reachable in \hat{M} . *Existential Abstraction* is a form of abstraction that preserves safety properties [CGL92].

Definition 3 (Existential Abstraction [CGL92]) *Given an abstraction function $\alpha : S \rightarrow \hat{S}$, a model $\hat{M} = (\hat{S}, \hat{S}_0, \hat{R})$ is called an Existential Abstraction of $M = (S, S_0, R)$ iff the following conditions hold:*

1. *The abstract model can make a transition from an abstract state \hat{s} to \hat{s}' iff there is a transition from s to s' in the concrete model and s is abstracted to \hat{s} and s' is abstracted to \hat{s}' :*

$$\begin{aligned} \forall \hat{s}, \hat{s}' \in (\hat{S} \times \hat{S}). \hat{R}(\hat{s}, \hat{s}') \iff \\ (\exists s, s' \in (S \times S). R(s, s') \wedge \alpha(s) = \hat{s} \wedge \alpha(s') = \hat{s}') \end{aligned} \quad (20.5)$$

2. *The set of abstract initial states \hat{S}_0 is simply the abstraction of the set of concrete initial states:*

$$\hat{S}_0 = \alpha(S_0) \quad (20.6)$$

The fact that existential abstraction is a conservative abstraction with respect to safety properties is formalized as follows:

Theorem 1 Let \hat{M} denote an existential abstraction of M , and let p denote a safety property. If p holds on \hat{M} , it also holds on M :

$$\hat{M} \models p \implies M \models p$$

Thus, for an existential abstraction \hat{M} and any program location l that is not reachable in the abstract model \hat{M} , we may safely conclude that it is also unreachable in the concrete model M . Note that the converse does not hold, i.e., there may be locations that are reachable in \hat{M} but not in M .

In program verification, the abstract transition relation \hat{R} is typically represented using a partitioning, similarly to the concrete transition relation R (see section 20.2). The abstract transition relation for program location $l \in \mathcal{L}$ is denoted by $\hat{R}_l(\hat{s}, \hat{s}')$:

$$\hat{R}(\hat{s}, \hat{s}') : \iff \bigwedge_{l \in \mathcal{L}} (s.l = l \longrightarrow \hat{R}_l(\hat{s}, \hat{s}')) \quad (20.7)$$

As the abstract program has the same control flow structure as the original program, the computation of \hat{R} follows the structure of the partitioning according to the program locations, i.e., \hat{R} is generated by computing \hat{R}_l from R_l for each location $l \in \mathcal{L}$ separately. In the following, we describe algorithms for this task.

20.5.3. Predicate Abstraction

There are various possible choices for an abstraction function α . *Predicate Abstraction* is one possible choice. It is one of the most popular and widely applied methods for systematic abstraction of programs. Predicate abstraction abstracts data by only keeping track of certain predicates on the data. The predicates are typically defined by Boolean expressions over the concrete program variables. Each predicate is then represented by a Boolean variable in the abstract program, while the original data variables are eliminated. The resulting program is called *Boolean Program*. Due to the smaller state-space of the Boolean program, Model Checking becomes feasible. A manual form of predicate abstraction was introduced by Graf and Saïdi [GS97]. An automated procedure to generate predicate abstractions was introduced by Colón and Uribe [CU98]. Predicate abstraction is promoted by the success of the SLAM project at Microsoft Research [BR00b]. The SLAM tool checks control-flow dominated properties of Windows device drivers.

The difficulty of predicate abstraction is to identify appropriate predicates, since they determine the accuracy of the abstract model. The predicates can be automatically inferred using *Counterexample-guided abstraction refinement* [Kur94, CGJ⁺00]: Abstract counterexamples that cannot be replayed in the concrete program serve as guide for refining the abstract model.

We postpone the problem of obtaining suitable predicates for now, and instead, focus on how to compute the abstract model once a set of predicates is given. We denote the set of Boolean values by $\mathbb{B} := \{\text{T}, \text{F}\}$. Let $\Pi := \{\pi_1, \dots, \pi_n\}$ denote the set of predicates. An abstract state \hat{s} consists of the program location and a valuation of the predicates, i.e., $\hat{S} = \mathcal{L} \times \mathbb{B}^n$. We denote the program location of an abstract state \hat{s} by $\hat{s}.l$, and the vector of predicates by $\hat{s}.\pi$. We

<pre> int main() { int i; ... L1: i=0; ... L2: while(even(i)) L3: i++; L4: ... return 0; } </pre>	<pre> void main() begin decl p1, p2; ... L1: p1,p2 := T,T; ... L2: while p2 do L3: p1,p2 := (p1 p2)?F:* ,!p2; od; L4: ... end </pre>
--	---

Figure 20.7. A C program and its corresponding Boolean program for the predicates $\pi_1 \iff i = 0$ and $\pi_2 \iff \text{even}(i)$. The predicate π_1 is represented by the program variable **p1**, while π_2 is represented by **p2**.

denote the value of predicate i by $\hat{s}.\pi_i$. The abstraction function $\alpha(s)$ maps a concrete state $s \in S$ to an abstract state $\hat{s} \in \hat{S}$:

$$\alpha(s) := \langle s.\ell, \pi_1(s), \dots, \pi_n(s) \rangle \quad (20.8)$$

We use Figure 20.7 as a running example to illustrate the concept of Boolean programs. It shows a small C program and its corresponding Boolean program. The syntax of Boolean programs follows that of most traditional sequential programming languages. Assume that the following set of predicates is given:

$$\begin{aligned} \pi_1 &\iff i = 0 \\ \pi_2 &\iff \text{even}(i) \end{aligned}$$

The predicate $\text{even}(i)$ holds iff i is an even number. The Boolean program uses the variables **p1** and **p2** to keep track of the value of these two predicates. The abstraction of the first assignment to the program variable **i** is straight-forward: both predicates hold after the execution of the statement. Boolean programs permit a parallel assignment to multiple variables, and thus, the pair **T, T** is assigned to the vector of the two predicates.

The Boolean program has the same control flow structure as the original program, and thus, the **while** loop is replicated. The abstraction of the condition is trivial, as the expression used as condition in the concrete program is available as predicate π_2 .

The abstraction of the **i++** statement inside the **while** loop is more complex: again, both predicates may change, and thus, a parallel assignment to **p1** and **p2** is performed. The new value of **p2** is the negated value of **p2**, as an even number becomes odd when incremented, and vice versa. The new value of **p1** depends on the current value of the predicates as follows:

- If $p1$ is \top , i is zero, we know that the new value of i is 1. Consequently, the new value of $p1$ must be F . Similarly, if $p2$ is \top , we can conclude that $i \neq -1$, the new value of i cannot be 0, and thus, the new value of $p1$ must be F .²
- If both $p1$ and $p2$ are F , we cannot make any conclusion about the new value of $p1$; the variable i may be -1, which yields \top as new value of $p1$, or a value that yields F , e.g., the value 3.

The second case above requires non-determinism in the Boolean program; i.e., the program may proceed into a state in which the new value of $p1$ is not determined. We use a ‘*’ to denote a non-deterministic value within the Boolean program.

In practice, Boolean programs are constructed algorithmically from the C program, that is, an algorithm computes \hat{R}_l given R_l and the set of predicates. \hat{R} is also called the *image* of the predicates Π over R .

Continuing our running example, consider the program statement $i++$ at $L3$ in the **while** loop, which has the most complex abstraction. Disregarding the control flow, this statement translates to the following concrete transition relation $R_l(s, s')$:

$$R_{L3}(s, s') \iff s'.i = s.i + 1$$

Again, assume that the set of predicates consists of $\pi_1 \iff i = 0$ and $\pi_2 \iff \text{even}(i)$. With $n = 2$ predicates, there are $(2^n) \cdot (2^n) = 16$ potential abstract transitions. A naïve way of computing \hat{R} is to enumerate the pairs \hat{s}, \hat{s}' and to check Eq. 20.5 for each such pair separately. As an example, the transition from $\hat{s} = \langle L3, \text{F}, \text{F} \rangle$ to $\hat{s}' = \langle L2, \text{F}, \text{F} \rangle$ corresponds to the following formula over concrete states:

$$\exists s, s'. \underbrace{\neg s.i = 0}_{\neg\pi_1} \wedge \underbrace{\neg \text{even}(s.i)}_{\neg\pi_2} \wedge \underbrace{s'.i = s.i + 1}_{R_{L3}} \wedge \underbrace{\neg s'.i = 0}_{\neg\pi'_1} \wedge \underbrace{\neg \text{even}(s'.i)}_{\neg\pi'_2} \quad (20.9)$$

Formula 20.9 can then be passed to a satisfiability checker as described in Sec. 20.3. The abstract transition is in \hat{R}_{L3} if and only if the formula is satisfiable. Continuing our example, the decision procedure will find Eq. 20.9 to be unsatisfiable, and thus, the particular abstract transition is not in \hat{R}_{L3} . Fig. 20.8 shows the abstract transitions for the program statement above, and one corresponding concrete transition (i.e., a satisfying assignment to Eq. 20.5) for each possible abstract transition.

The procedure described above results in the ‘best’ possible, i.e., the most precise abstract transition relation for the given predicates. The drawback of this approach is that the run-time of the abstraction procedure is exponential in the number of predicates. The only predicate abstraction framework implementing this approach is MAGIC [CCG⁺04].

20.5.4. Successive Approximation of \hat{R}

Most implementations of predicate abstraction approximate \hat{R} iteratively, in order to avoid the exponential number of calls to the theorem prover. This is done

²Note that the ANSI-C standard does not provide any guarantees in case of overflow of signed integers; however, we can argue that an even number does not overflow if incremented.

Abstract Transition				Concrete Transition	
$\hat{s}.\pi_1$	$\hat{s}.\pi_2$	$\hat{s}'.\pi_1$	$\hat{s}'.\pi_2$	s	s'
F	F	F	T	$s.i = 1$	$s'.i = 2$
F	F	T	T	$s.i = -1$	$s'.i = 0$
F	T	F	F	$s.i = 2$	$s'.i = 3$
T	T	F	F	$s.i = 0$	$s'.i = 1$

Figure 20.8. Example for existential abstraction: Let the concrete transition relation $R_{L3}(s, s')$ be $s'.i = s.i + 1$ and let $\pi_1 \iff i = 0$ and $\pi_2 \iff \text{even}(i)$. The table lists the transitions in \hat{R}_{L3} and an example for a corresponding concrete transition.

by computing a series of over-approximations \hat{R}' of \hat{R} , and is also called *Lazy Abstraction*. The resulting abstract model is weaker as it permits more paths than the model obtained using \hat{R} . Note that the claim in Theorem 1 still holds, as any over-approximation of \hat{R} fulfills the requirements of an existential abstraction. An example of such a procedure is the abstraction component C2BP of the SLAM Model Checker [BPR01, BMMR01].

Predicate Partitioning One technique to compute an over-approximation of \hat{R} is *Predicate Partitioning*. It was introduced in the context of predicate abstraction for Verilog designs by Jain et al. [JKSC05]. As a first step, the set of predicates is partitioned into a number of clusters.

We denote the clusters by $\mathcal{C}_1, \dots, \mathcal{C}_k$. For the purposes of this partitioning, the predicates over the current state s are distinguished from the predicates in the next state s' : Let $\Pi = \{\pi_1, \dots, \pi_n\}$ denote the set of predicates over s , and $\Pi' = \{\pi'_1, \dots, \pi'_n\}$ denote the same predicates over s' . A cluster \mathcal{C}_i is a non-empty subset of $\Pi \dot{\cup} \Pi'$. The union of the clusters is not required to be disjoint or to cover all predicates.

Given a partitioning \mathcal{P} , we denote the existential abstraction of R_l with respect to \mathcal{P} by $\hat{R}_l^{\mathcal{P}}$. It is defined by means of k separate abstract transition relations $\hat{R}_l(1), \dots, \hat{R}_l(k)$, where $\hat{R}_l(j)$ is the precise existential abstraction of R_l with respect to the predicates in the cluster \mathcal{C}_j :

$$(\hat{s}, \hat{s}') \in \hat{R}_l(j) \iff \exists s, s'. R_l(s, s') \wedge \bigwedge_{\pi_i \in \mathcal{C}_j} \hat{s}.\pi_i = \pi_i(s) \wedge \bigwedge_{\pi'_i \in \mathcal{C}_j} \hat{s}'.\pi_i = \pi_i(s') \quad (20.10)$$

The abstract transition relations $\hat{R}_l(j)$ are conjoined to form $\hat{R}_l^{\mathcal{P}}$:

$$\hat{R}_l^{\mathcal{P}} := \bigwedge_{j=1}^k \hat{R}_l(j) \quad (20.11)$$

It is easy to see that $\hat{R}_l \subseteq \hat{R}_l^{\mathcal{P}}$, i.e., $\hat{R}_l^{\mathcal{P}}$ is an over-approximation of \hat{R}_l . The abstract transition relation for each cluster can be computed by using all-SAT as suggested in [CKSY04] or by any other precise abstraction method, e.g., enumeration as described above, or by means of BDDs.

The choice of clusters is typically done by a syntactic analysis of \hat{R}_l and the predicates, e.g., by analyzing the variables that are mentioned in the predicates

and what variables are affected by a transition. A possible strategy is to put predicates that share variables into common clusters.

As a result of the over-approximation \hat{R} , we may obtain an abstract counterexample $\hat{s}_1, \dots, \hat{s}_m$ that contains abstract transitions $\langle \hat{s}_i, \hat{s}_{i+1} \rangle \in \hat{R}'$ for which no concrete counterpart exists, i.e., $\langle \hat{s}_i, \hat{s}_{i+1} \rangle \notin \hat{R}$. Such a transition is called a *spurious transition*. In such a case, \hat{R}' is strengthened by removing some of the transitions. We defer the discussion of how to refine \hat{R}' to section 20.5.7.

20.5.5. Verification of the Abstract Model

Once the Boolean program is constructed, it is passed to a model checker. The purpose of this phase is to determine if there are paths that can reach an error location. Such a path is called an *abstract counterexample*, and is formalized as follows:

Definition 4 (Abstract Counterexample) *An abstract counterexample \hat{t} is a witness for the reachability of an error state \hat{s}_n (where $\hat{s}_n.l \in \mathcal{L}_E$) in the abstract model \hat{M} . Analogously to concrete counterexamples (Def. 2), an abstract counterexample is provided by means of a finite execution trace $\hat{s}_0, \hat{s}_1, \dots, \hat{s}_n$, for which $\bigwedge_{0 \leq i < n} \hat{R}(\hat{s}_i, \hat{s}_{i+1})$ and $\hat{s}_0 \in \hat{S}_0$ holds.*

As mentioned above, the Boolean program has exactly the same control flow structure as its corresponding original program, including (possibly recursive) function calls. There is a wide range of model checking techniques available for the analysis of such abstract models, depending on the set of control flow constructs used by the program.

While the reachability problem for sequential C programs is undecidable, it is decidable for sequential Boolean programs, despite the presence of a potentially unbounded call stack [Büc64]. Since only topmost variables of the stack are visible at each program point, the successor state of a transition is exclusively determined by them and the global variables. Once the return value of a procedure call for certain input values (i.e., global variables and actual parameters) is determined, a second evaluation of the function call with the same arguments is redundant [SP81]. This can be exploited by storing input-output pairs for each procedure (such a pair is called a *summary edge* [BR00a, FWW97]). The reachable states are then computed using a fixed point computation.

All existing model checkers for Boolean programs use a symbolic representation for states: Instead of explicitly enumerating all possible states, Boolean formulas are used to represent sets of states. Binary Decision Diagrams (BDDs) are a particularly well suited representation for Boolean formulas used in fixed point computations, since the computation of the set of successor states can be performed easily [Bry86]. However, BDDs do not scale for a growing number of variables (the upper limit is typically reached with about 100 variables). The number of variables a SAT-solver can handle is several magnitudes higher, but the SAT-based approach requires Quantified Boolean formulas (QBF) for the detection of fixed points [CKS05b]. QBF is a classical PSPACE-complete problem, and is discussed in Part 2, Chapter 29. In practice, QBF-solvers are not nearly as scalable as propositional SAT-solvers. BOPPO is an implementation of a model

checker for concurrent Boolean programs that is based on propositional SAT for path checks, and on QBF for the fixed point detection [CKS07].

If no error state is reachable, then Theorem 1 guarantees that there is also no counterexample in M , and the verification algorithm terminates. In case an abstract counterexample is found, we cannot simply conclude that there is a corresponding counterexample for the original program, as the converse of Theorem 1 does not hold. The abstract counterexample has to be *simulated* on the original program. We describe this process in the next section.

20.5.6. Simulation

As explained in the previous section, not every abstract counterexample has a corresponding concrete counterpart. Therefore, the feasibility of an abstract counterexample must be verified before it is reported to the user. Infeasible counterexamples are eliminated by refining the abstraction, i.e., the approach does not report false negatives.

The easiest way to check if there is a concrete counterexample in M that corresponds to a given abstract counterexample \hat{t} is to perform *symbolic simulation* along the path given by \hat{t} . Recall that \hat{t} is a sequence $\hat{s}_0, \hat{s}_1, \dots, \hat{s}_n$ of abstract states, each of which contains a program location $\hat{s}_i.\ell$. We use $l(i) := \hat{s}_i.\ell$ as a short-hand for these program locations.

In order to check if there is a path in M that follows $l(1), \dots, l(n)$, we iteratively build a formula F_n as follows:

$$\begin{aligned} F_0(s_0) &:= \top \\ F_i(s_i) &:= F_{i-1} \wedge R_{l(i-1)}(s_{i-1}, s_i) \end{aligned} \quad (20.12)$$

The construction of F_n corresponds to the symbolic execution of the transition relation R along the program locations given by \hat{t} . It can be passed to a SAT-based decision procedure as described in Sec. 20.3. If it is found to be satisfiable, a concrete counterexample can be extracted from the satisfying assignment.

Note that such a counterexample does not necessarily comply with the values of the predicates given by \hat{t} , as F_n only follows the program locations. The constraints in Eq. 20.12 are sufficient, since we are satisfied with *any* feasible concrete counterexample.

The tools SLAM and BLAST rely on the First-Order logic theorem provers ZAPATO [BCLZ04] and Simplify [DNS03], respectively, for the simulation of abstract counterexamples. There is no guarantee that these decision procedures provide a definite answer, since the counterexample trace may contain transitions that make use of multiplication, division, or bit-wise operators. In these cases, false negatives are reported. Furthermore, bugs that originate from an integer overflow may go undetected, as neither SLAM nor BLAST provide an accurate axiomatization of bit-vector arithmetic.

A SAT-based procedure as described in Sec. 20.3 mends these problems. SAT solvers provide a witness for satisfiability, which can be mapped back to a valuation to the variables of the feasible counterexample. This additional information makes counterexamples significantly easier to understand for programmers. It should be mentioned that SLAM is not restricted to its default theorem prover

ZAPATO: The SAT-based theorem prover Cogent, which supports bit-vectors and provides countermodels, has been integrated into an experimental version of SLAM [CKS05a].

What if Eq. 20.12 is unsatisfiable? In this case, there is no concrete counterexample that follows the locations $l(1), \dots, l(n)$, and the abstract counterexample is *spurious*. There may be two reasons for a spurious counterexample: we either lack some important predicate, or the counterexample is caused by our over-approximation of \hat{R} , as described in Sec. 20.5.4. In either case, we have to *refine* our abstraction, which is described next.

20.5.7. Refinement

The accuracy of the abstract model depends on the set of predicates. Coarser abstractions contain more “undesirable” traces. There are two kinds of artifacts introduced by the abstraction mechanism: (a) An insufficient set of predicates results in *spurious traces*, and (b) over-approximation of the abstract transition function yields *spurious transitions*. We first describe how to refine spurious transitions, and then present predicate refinement techniques.

Spurious Transitions Predicate abstraction techniques like lazy abstraction, the Cartesian approximation [BPR03] or predicate partitioning (see Sec. 20.5.4) trade precision for performance. Therefore, important correlations between predicates may be lost. Ball et al. present a technique that refines these correlations only in the cases in which they are relevant [BCDR04]. The approach is based on Das and Dill’s algorithm, which refines an abstract transition relation by eliminating *spurious transitions* by means of the predicates already available [DD01].

An abstract transition in a counterexample is spurious if it does not have a corresponding concrete transition (Eq. 20.5). For each transition, an abstract counterexample provides the abstract states before (\hat{s}) and after (\hat{s}') that transition. One way of checking if an abstract error trace contains spurious transitions is to verify Eq. 20.5 for each of the steps in the trace individually using a decision procedure for bit-vector arithmetic. If Eq. 20.5 does not hold for a pair $\langle \hat{s}, \hat{s}' \rangle$, then the corresponding abstract transition is spurious. The abstract transition relation is then refined by ruling out the combination $\langle \hat{s}, \hat{s}' \rangle$ of states.

Example 2 *As an example, consider a program with an integer variable x , and let the instruction at the program location that we check be $x++$. The abstraction is based on the three predicates $x = 1$, $x = 2$, and $x = 3$, called π_1 , π_2 , and π_3 , respectively, and is initially just unconstrained:*

$$p1, p2, p3 := *, *, *;$$

The transition we consider is $\langle T, F, F \rangle$ to $\langle T, F, F \rangle$, which corresponds to the following instance of Eq. 20.5:

$$\exists x, x'. x = 1 \wedge x \neq 2 \wedge x \neq 3 \wedge x' = x + 1 \wedge x' = 1 \wedge x' \neq 2 \wedge x' \neq 3 \quad (20.13)$$

As there is no model for this formula, the transition is spurious. One way to refine the abstract model is to add a constraint to \hat{R} that rules out this particu-

lar transition. Boolean programs offer a constrained assignment for exactly this purpose³:

```
p1, p2, p3 := *,*,* constrain !(p1 & !p2 & !p3 & 'p1 & 'p2 & 'p3);
```

Obviously, ruling out spurious transitions one-by-one as in the example above might require an exponential number of refinement iterations. One way of eliminating multiple spurious transitions with a single iteration is to examine the *unsatisfiable core* of Eq. 20.5. The unsatisfiable core contains only those clauses that contribute to the proof of unsatisfiability. As SAT-solvers attempt to construct small proofs, and thus, details of the input formula that are irrelevant are usually not in the core. Algorithms for the construction of the smallest core are described in Part 1, Chapter 14. It is easy to see that the predicates that are irrelevant to the proof of unsatisfiability can be removed from the constraint that is added.

Example 3 *Continuing our example, observe that the following subset of the constraints of Eq. 20.13 is already unsatisfiable:*

$$x = 1 \wedge x' = x + 1 \wedge x' \neq 2 \quad (20.14)$$

Only two of the literals of the constrain clause in Example 2 are actually used, and we therefore may replace it with the following much stronger statement:

```
p1, p2, p3 := *,*,* constrain !(p1 & !'p2);
```

While this does not rule out all spurious transitions, it may result in an exponential reduction of the number of refinement iterations.

In SLAM, the abstract counterexample is examined for spurious transitions whenever a set of refinement predicates fails to eliminate the spurious counterexample it is derived from. The corresponding SLAM component is called CON-STRAIN [BCDR04]. CON-STRAIN uses the decision procedure ZAPATO [BCLZ04] in order to decide if a given abstract transition is spurious. SATABS performs the check for spurious transitions before the predicate refinement is done: in many cases, this prevents that unnecessary predicates are added. BLAST only performs a very simplistic initial abstraction.

Spurious Traces Even if an abstract counterexample is free of spurious transitions, the combination of the transitions together might not have a corresponding path in the original program. Spurious traces can be eliminated by adding additional predicates.

All refinement techniques rely on heuristics to extract a small subset of predicates that explain the infeasibility of the counterexample. As example, consider the programs in Fig. 20.9. The assertion in the program is automatically generated to assure that the array index operation in the C program does not exceed the upper bound. Assume that our initial set of predicate consists of those mentioned in the assertion, i.e., we have $i < 10$ as only predicate. The Boolean

³Note that Boolean programs use $'p$ to denote the next-state value of p , instead of the more commonly used notation p' .

<pre> int a[10], i; ... L1: i=0; L2: while(i!=10) { L3: assert(i<10); L4: a[i]=0; L5: i++; } </pre>	<pre> decl p1; ... L1: p1 := T; L2: while * do L3: assert(p1); L4: skip; L5: p1 := *; od; </pre>
---	---

Figure 20.9. The C program on the left initializes an array with zero by means of an index variable. It contains an automatically generated assertion that checks that the index stays within the range of the array. The Boolean program on the right uses the predicate $i < 10$ from the property.

program on the right hand side of the figure is trivial, as only two instructions do actually affect the predicate. For the sake of this example, we assume that \hat{R} is computed precisely, and thus, $p1$ is assigned T in $L1$ and non-deterministically in $L5$.

The verification phase reveals that there is a counterexample in the Boolean program. The first iteration through the loop may reach a state in which $p1$ is F , and this violates the assertion at the beginning of the second iteration. The simulation phase could build the following formula to check if that path is feasible in the original program (we omit the values of the array for brevity):

$$\underbrace{i_1 = 0}_{L1} \wedge \underbrace{i_1 \neq 10}_{L2} \wedge \underbrace{i_2 = i_1 + 1}_{L5} \wedge \underbrace{i_2 \neq 10}_{L2} \wedge \underbrace{i_2 \geq 10}_{L3} \quad (20.15)$$

As Eq. 20.15 is unsatisfiable, the counterexample is spurious. As we did not use an over-approximation of \hat{R} , no spurious transitions can be found in the counterexample, and thus, we must refine the set of predicates. One way of refining the predicates is to compute the *strongest post-condition* along the path given by the counterexample, and add the predicates that are found in this process.

The strongest post-condition of $L1$ is $i = 0$, and the strongest post-condition of $i = 0$ with respect to $L5$ is $i = 1$. Adding the predicates $i = 0$ and $i = 1$ eliminates the spurious counterexample we have obtained. We name the new predicates $p2$ and $p3$, respectively. Fig. 20.10 shows the Boolean program after adding these predicates, where ‘ $p2 \rightarrow p3$ ’ denotes the logical implication of $p3$ by $p2$. The constrained assignment at $L5$ could be generated by the procedure described above, and is strong enough to rule out the spurious counterexample.

However, the refined model still contains a counterexample: after iterating through the loop twice, the assertion in the Boolean program is violated. This corresponds to the concrete state in which $i = 2$. Due to lack of appropriate predicates, the abstraction is unable to distinguish this state from states that

```

    decl p1, p2, p3;
    ...
L1: p1 := T, T, F;

L2: while * do
L3:   assert (p1);
L4:   skip;
L5:   p1, p2, p3 := *, *, * constrain p2 -> p3 & p3 -> p1;
    od;

```

Figure 20.10. A refined abstraction of the C program in Fig. 20.9.

violate $i < 10$. Continuing the refinement using strongest post-conditions eventually eliminates all spurious counterexamples, and thus, proves the property. However, this requires already nine refinement iterations for our little example. The reason for this is that a predicate for an essential part of the loop invariant is missing: given the predicate $i \geq 0$, predicate abstraction is able to prove the property without additional refinement iteration.⁴

Various techniques to speed up the refinement and the simulation have been proposed. *Path slicing* eliminates instructions from the counterexample that do not contribute to the violation of the property in question [JM05]. *Loop detection* is a technique that considers an arbitrary number of iterations of potential loops in the counterexample in a single simulation step [KW06]. Jain et al. suggest to add statically inferred program invariants in order to speed up the refinement [JIG⁺06].

Craig interpolation [Cra57] is an alternative approach that aims at finding parsimonious predicates that are just sufficient for eliminating a given spurious counterexample [HJMM04]. If augmented with techniques that force a generalization of the interpolants, the predicates needed for representing the loop invariant are frequently obtained [JM06]. Craig interpolants for propositional formulas are explained in Part 2, Chapter 18.

Predicate abstraction is typically restricted to safety properties. However, predicate abstraction can be combined with algorithms for the computation of *ranking functions* in order to argue termination of program fragments [CPR05]. As an example, this permits showing that a lock that is acquired is eventually released.

20.6. Conclusion

BMC is the best technique to find shallow bugs. It supports the widest range of program constructions. BMC does not rely on built-in knowledge about the data structures the program maintains. On the other hand, completeness is only obtainable on very 'shallow' programs, i.e., programs without deep loops.

⁴Recall that the addition may overflow!

Predicate abstraction is known to be powerful when applied to light-weight control-flow dominated property. As it removes detail that is irrelevant to the property from the program, it can scale to larger programs. Considerations with respect to the depth of the model are not required. Predicate abstraction has to rely on additional knowledge about the data structures that the property depends on.

In both cases, SAT-based decision procedures enable the verification engine to reason about bit-vector logic, and to treat variable overflow in a sound manner. The countermodels generated by typical SAT-solvers provide an important debugging aid to the programmer.

References

- [AMP06] A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. In *Model Checking and Software Verification (SPIN)*, volume 3925 of *LNCS*, pages 146–162. Springer, 2006.
- [AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Principles of Programming Languages (POPL)*, pages 1–11. ACM, 1988.
- [BCDR04] T. Ball, B. Cook, S. Das, and S. K. Rajamani. Refining approximations in software predicate abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *LNCS*. Springer, 2004.
- [BCLZ04] T. Ball, B. Cook, S. K. Lahiri, and L. Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In *Computer Aided Verification (CAV)*, volume 3114 of *LNCS*. Springer, 2004.
- [BCRZ99] A. Biere, E. M. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a Power PC microprocessor using symbolic model checking without BDDs. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification (CAV)*, volume 1633 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 1999.
- [BDL98] C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for bit-vector arithmetic. In *Design Automation Conference (DAC)*, pages 522–527. ACM, June 1998.
- [BGD05] S. Berezin, V. Ganesh, and D. Dill. A decision procedure for fixed-width bit-vectors. Technical report, Computer Science Department, Stanford University, 2005.
- [BKO⁺07] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. Brady. Deciding bit-vector arithmetic with abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4424 of *LNCS*. Springer, 2007.
- [BMMR01] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI)*, pages 203–213. ACM, 2001.
- [BPR01] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and Cartesian abstractions for model checking C programs. In *Tools and Algorithms*

- for the Construction and Analysis of Systems (TACAS), volume 2031 of LNCS, pages 268–283. Springer, 2001.
- [BPR03] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstraction for model checking C programs. *Software Tools for Technology Transfer (STTT)*, 5(1):49–58, 2003.
- [BR00a] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Model Checking and Software Verification (SPIN)*, volume 1885 of LNCS, pages 113–130. Springer, 2000.
- [BR00b] T. Ball and S. K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report 2000-14, Microsoft Research, February 2000.
- [Bry86] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
- [Büc64] J. R. Büchi. Regular canonical systems. *Archive for Mathematical Logic*, 6(3-4):91, April 1964.
- [CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL)*, pages 238–252. ACM, 1977.
- [CCG⁺04] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering (TSE)*, pages 388–402, June 2004.
- [CGJ⁺00] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification (CAV)*, pages 154–169. Springer, 2000.
- [CGL92] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Principles of Programming Languages (POPL)*, pages 343–354. ACM, 1992.
- [CGP⁺06] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *Computer and Communications Security (CCS)*, pages 322–335. ACM, 2006.
- [CHR00] D. W. Currie, A. J. Hu, and S. P. Rajan. Automatic formal verification of DSP software. In *Design Automation Conference (DAC)*, pages 130–135. ACM, 2000.
- [CKL04] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of LNCS, pages 168–176. Springer, 2004.
- [CKS05a] B. Cook, D. Kroening, and N. Sharygina. Cogent: Accurate theorem proving for program verification. In *Computer Aided Verification (CAV)*, volume 3576 of LNCS, pages 296–300. Springer, 2005.
- [CKS05b] B. Cook, D. Kroening, and N. Sharygina. Symbolic model checking for asynchronous Boolean programs. In *Model Checking and Software Verification (SPIN)*, pages 75–90. Springer, 2005.
- [CKS07] B. Cook, D. Kroening, and N. Sharygina. Verification of Boolean programs with unbounded thread creation. *Theoretical Computer Science (TCS)*, 388:227–242, 2007.

- [CKSY04] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design (FMSD)*, 25:105–127, September–November 2004.
- [CMR97] D. Cyrluk, M. O. Möller, and H. Rueß. An efficient decision procedure for the theory of fixed-sized bit-vectors. In *Computer Aided Verification (CAV)*, LNCS, pages 60–71. Springer, 1997.
- [CPR05] B. Cook, A. Podelski, and A. Rybalchenko. Abstraction refinement for termination. In C. Hankin and I. Siveroni, editors, *Static Analysis (SAS)*, volume 3672 of *Lecture Notes in Computer Science*, pages 87–101. Springer, 2005.
- [Cra57] W. Craig. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *Journal of Symbolic Logic*, 22(3):250–268, 1957.
- [CU98] M. A. Colón and T. E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Computer Aided Verification (CAV)*, volume 1427 of *LNCS*, pages 293–304. Springer, 1998.
- [DD01] S. Das and D. L. Dill. Successive approximation of abstract transition relations. In *Logic in Computer Science (LICS)*, pages 51–60. IEEE, 2001.
- [DdM06] B. Dutertre and L. de Moura. The Yices SMT solver. Available at <http://yices.cs1.sri.com/tool-paper.pdf>, September 2006.
- [DKW08] V. D’Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7):1165–1178, July 2008.
- [DNS03] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, January 2003.
- [FLL⁺02] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Language Design and Implementation (PLDI)*, pages 234–245. ACM, 2002.
- [FWW97] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. In *Verification of Infinite State Systems (INFINITY)*, volume 9 of *ENTCS*. Elsevier, 1997.
- [GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verification (CAV)*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.
- [HJMM04] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *Principles of Programming Languages (POPL)*, pages 232–244. ACM, 2004.
- [ISGG05] F. Ivancic, I. Shlyakhter, A. Gupta, and M. K. Ganai. Model checking C programs using F-SOFT. In *International Conference on Computer Design (ICCD)*, pages 297–308. IEEE, 2005.
- [Jac06] D. Jackson. The Alloy analyzer, 2006. <http://alloy.mit.edu/>.
- [JIG⁺06] H. Jain, F. Ivancic, A. Gupta, I. Shlyakhter, and C. Wang. Using statically computed invariants inside the predicate abstraction and

- refinement loop. In *Computer Aided Verification (CAV)*, volume 4144 of *LNCS*, pages 137–151. Springer, 2006.
- [JKSC05] H. Jain, D. Kroening, N. Sharygina, and E. M. Clarke. Word level predicate abstraction and refinement for verifying RTL Verilog. In *Design Automation Conference (DAC)*, pages 445–450. ACM, 2005.
- [JM05] R. Jhala and R. Majumdar. Path slicing. In *Programming Language Design and Implementation (PLDI)*, pages 38–47. ACM, 2005.
- [JM06] R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3920 of *Lecture Notes in Computer Science*, pages 459–473. Springer, 2006.
- [KCY03] D. Kroening, E. M. Clarke, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Design Automation Conference (DAC)*, pages 368–371. ACM, 2003.
- [KS03] D. Kroening and O. Strichman. Efficient computation of recurrence diameters. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 2575 of *LNCS*, pages 298–309. Springer, 2003.
- [Kur94] R. P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994.
- [KW06] D. Kroening and G. Weissenbacher. Counterexamples with loops for predicate abstraction. In *Computer Aided Verification (CAV)*, volume 4144 of *LNCS*, pages 152–165. Springer, 2006.
- [MMZ⁺01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference (DAC)*, pages 530–535. ACM, 2001.
- [RG05] I. Rabinovitz and O. Grumberg. Bounded model checking of concurrent programs. In *Computer Aided Verification (CAV)*, volume 3576 of *LNCS*, pages 82–97. Springer, 2005.
- [SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.
- [WSK05] M. Wedler, D. Stoffel, and W. Kunz. Normalization at the arithmetic bit level. In *Design Automation Conference (DAC)*, pages 457–462. ACM, 2005.
- [XA05] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *Principles of Programming Languages (POPL)*, pages 351–363. ACM, 2005.
- [YST⁺06] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. R. Engler. Automatically generating malicious disks using symbolic execution. In *IEEE Symposium on Security and Privacy (S&P)*, pages 243–257. IEEE, 2006.

Chapter 21

Combinatorial Designs by SAT Solvers¹

Hantao Zhang²

21.1. Introduction

The theory of combinatorial designs has extensive interactions with numerous areas of mathematics. Group theory, graph theory, number theory, finite geometry, and linear algebra all have close connections with design theory. Design theory also has applications in numerous disciplines: experimental design, coding theory, cryptography, and many areas of computer science such as data structures, algorithms, and AI. Needless to say, design theory provides a rich source of structured, parameterized families of hard propositional theories, and provides numerous application problems for SAT solvers. As the language of the first NP-complete problem, the propositional logic is expressive enough to encode many problems in design theory. Dramatic improvements in the efficiency of SAT solvers encourage people to attack design theory problems simply by encoding problems as propositional formulas, and then searching for their models using general purpose off-the-shelf SAT solvers. In this chapter, we will present several case studies of such attempts.

Most mathematicians working in the field of design theory have access to modern computers and have worked on various special-purpose software tools for problems in this field [Ce96]. Some of these tools are incomplete and are based on the special knowledge of the field; some of them are complete, based on exhaustive backtrack search algorithms.

Since 1990, general-purpose model generation tools, including SAT solvers, are used extensively to find finite structures of many design theory problems [Zha97]. The so-called general-purpose model generation tools (or simply model generators) are the software tools to solve constraint satisfaction problems in AI, to prove theorems in finite domains, or to produce counterexamples to false conjectures.

While both special-purpose complete tools and general-purpose model generators rely on exhaustive search, there are fundamental differences. For the latter,

¹This chapter is dedicated to Dr. Masayuki Fujita, who passed away while traveling in China, in 2005.

²Partially supported by the National Science Foundation under Grants CCR-0604205.

every problem has a uniform internal representation (i.e., propositional clauses for SAT solvers). This uniform representation may introduce redundancies and inefficiencies. However, since a single search engine is used for all the problems, any improvement to the search engine is significant. Moreover, we have an accumulated knowledge of three decades on how to make such a uniform search engine efficient. Using general-purpose model generators to solve combinatorial design theory problems deserves the attention for at least two reasons.

- It is much easier to specify a problem for a general-purpose model generator than to write a special program to solve it. Similarly, fine-tuning a specification is much easier than fine-tuning a special program.
- General-purpose model generators can provide competitive and complementary results for certain combinatorial design theory problems. Evidences will be provided in the following sections.

It appears that the first report of attacking design theory problems using a general-purpose model generator came out in 1991, when Jian Zhang solved a previously-open quasigroup problem using an early version of FALCON [Zha96] and reported this result to the community of Automated Reasoning [ZW91]. FALCON is a general-purpose model generation software based on backtracking search. In 1992, Masayuki Fujita of ICOT, Tokyo, applied their model generation based theorem prover, MGTP [HFKS02], to some quasigroup problems and solved a dozen of the previously-open cases. In the same year, John Slaney at Australian National University applied his general-purpose model generator called FINDER [Sla94] to the quasigroup problems. In addition to confirm Fujita's results, he also reported results for four previously-open problems. These results were reported at IJCAI in 1993 in their award-winning paper [FSB93].

Around the same time, there was a renewed interests on high performance SAT solvers. The quasigroup problems provided a significant spur to the independent developments of SAT solvers by Mark Stickel, William McCune, the author of the present chapter, among others. Stickel's DDPP and LDPP, the author's SATO and McCune's MACE were used to solve several dozen cases of previously-open quasigroup problems [McC94]. It turned out that SAT solvers performed surprisingly well on the quasigroup problems, providing an excellent example that general-purpose search tools may beat special-purpose search tools for certain design theory problems. Many previously-open cases of quasigroups with specified properties were first solved by these model generators. It shows that SAT solvers are very competitive against other general-purpose model generators as well as against special-purpose software tools for many combinatorial design theory problems. In the earlier stage of our study of Latin square problems, the author wrote two special-purpose programs. After observing that these two programs could not do better than SATO, the author has not written any special-purpose search programs since then.

In this chapter, we will present several combinatorial design problems solved by general-purpose model generators. Many of these problems have been collected at CSPLIB [GW99], a problem library for constraints, initially created by Ian Gent and Toby Walsh. Since all the finite model problems can be encoded in SAT, SAT solvers can be regarded as a special kind of general-purpose model

generators.

A typical SAT-based model generator consists of three components:

- an *encoder* which transforms a given problem into a propositional formula (in most cases, a set of clauses) such that the original problem has a model if and only if the formula is satisfiable;
- a SAT solver which decides if the clauses obtained from the encoder are satisfiable;
- a *decoder* which interprets the outcome of the SAT solver.

A software program which combines the above three components into one with a general-purpose input language is called *Mace-style* model generator, named after McCune's Mace 1.0 system.³

21.2. Combinatorial Design Problems

In this chapter, we present the following combinatorial design problems.

- Quasigroups and Latin squares,
- Ramsey numbers,
- Van der Waerden numbers,
- Covering arrays,
- Orthogonal arrays,
- Covering array numbers of small strength,
- Steiner systems,
- Mendelsohn designs, and
- Magic squares

Along the presentation, we show what are the problems solved by model generators and what are the remaining open problems. The issues of building an effective and efficient encoders for SAT solvers will be addressed near the end of this chapter.

21.2.1. Quasigroup Problems

The information on quasigroup and its related structures provided in this chapter is mainly drawn from a survey paper by Bennett and Zhu [BZ92] and the CRC Handbook of Combinatorial Designs [Ce96]; the interested reader may refer to that work for more information on quasigroup, their importance in design theory, and some other related applications.

Given a set S , a *Latin square indexed by S* is an $|S| \times |S|$ matrix such that each row and each column of the matrix is a permutation of the elements in S . $|S|$ is called the *order* of the Latin square.

From the algebraic point of view, a Latin square indexed by S defines a binary operator $*$ over the domain S and the Latin square is the “multiplication table” of $*$. The pair $(S, *)$ is a special *cancellative groupoid* in algebra, called *quasigroup*. Quasigroup is often used as a synonym of Latin square in the literature. By

³Before Mace 1.0, the less-known ModGen system [KZ94] has the same functionality as Mace 1.0.

abuse of notation, throughout this chapter, we use $(S, *)$ to denote a Latin square indexed by S . Latin squares have rich connection with many other fields of design theory. For instance, a Latin square of size n is equivalent to a transversal design of index one, a $(3, n)$ -net, an orthogonal array of strength two and index one, a 1-factorization of the complete bipartite graph $K_{n,n}$, an edge-partition of the complete tripartite graph $K_{n,n,n}$ into triangles, a set of n^2 mutually non-attacking rooks on a $n \times n \times n$ board, and a single error-detecting code of word length 3, with n^2 words from an n -symbol alphabet. Moreover, the underlying structure of Latin squares has many similarity comparing to many real-world applications, such as scheduling and time-tabling, experimental design, and wavelength routing in fiber optics networks [KRS99].

Without loss of generality, let S be $\{0, 1, \dots, (n-1)\}$. The following formulas, which are first-order logic clauses, specify a quasigroup $(S, *)$, or equivalently, a Latin square of order n . For all elements $x, y, u, w \in S$,

$$x * u = y, x * w = y \Rightarrow u = w \quad : \text{ the left-cancellation law} \quad (21.1)$$

$$u * x = y, w * x = y \Rightarrow u = w \quad : \text{ the right-cancellation law} \quad (21.2)$$

$$x * y = u, x * y = w \Rightarrow u = w \quad : \text{ the unique-image property} \quad (21.3)$$

$$(x * y = 0) \vee \dots \vee (x * y = (n-1)) \quad : \text{ the closure property} \quad (21.4)$$

The left-cancellation law states that no symbol appears in any column more than once and the right-cancellation law states that no symbol appears in any row more than once. The unique-image property states that each cell of the square can hold at most one symbol.

It is straightforward to convert the above formulas into the propositional clauses if we represent each $a * b = c$ by a propositional variable $p_{a,b,c}$. The number of propositional clauses generated from such a first-order clause is n^k , where n is the order of the Latin square to be searched and k is the number of free variables in the first-order clause. This is true for transforming flattened first-order clauses into propositional clauses.

In addition to the above clauses, Slaney, Fujita and Stickel [FSB93] have used the following two clauses, which are valid consequences of the above clauses, in their model generators:

$$(x * 0 = y) \vee \dots \vee (x * (n-1) = y) \quad : \text{ the closure property (a)} \quad (21.5)$$

$$(0 * x = y) \vee \dots \vee ((n-1) * x = y) \quad : \text{ the closure property (b)} \quad (21.6)$$

Another justification for the above two clauses is that the binary representation of a Latin square can be viewed as a cube, where the dimensions are the row, the column, and the symbol. Under this view, any set of variables determined by holding two of the dimensions fixed must contain exactly one true variable in the third dimension [GS02]. The effect of these two-clause schemes to the search time varies. For small order Latin squares, the effect is sometimes negative. When these two clauses are used together with the isomorphism-cutting clause (see section 21.3.2), the run time can be faster by a factor greater than 2. Carla Gomes and David Shmoys also found that these two clauses are very useful for the SAT-based approach to solve the quasigroup completion problem (see [GS02] and the next subsection for details).

21.2.1.1. Completion of a Partial Latin Square

An $n \times n$ matrix L (i.e., of order n) with cells that are either empty or contain exactly one symbol of S is a *partial* Latin square if no symbol occurs more than once in any row or column. If the first k rows of a partial Latin square ($k \leq n$) are all filled and the remaining cells are all empty, then L is a $k \times n$ *Latin rectangle*. A partial Latin square P of order n is *embedded* in a Latin square L of order t if the upper left corner of L agrees with P , where $n \leq t$.

Theorem 21.2.1. [Ce96]

(a) *A partial Latin square of order n with at most $n - 1$ filled cells can always be completed to a Latin square of size n .*

(b) *A $k \times n$ Latin rectangle, $k < n$, can always be completed to a Latin square of order n .*

(c) *A partial Latin square of order n can be embedded into a Latin square of order t for every $t \geq 2n$.*

A well-known example of the quasigroup completion problem is perhaps the Sudoku puzzle, which appears on many newspapers and magazines. The most popular form of Sudoku has a 9×9 grid made up of nine 3×3 subgrids called “regions”. In addition to the constraints that every row and every column is a permutation of 1 through 9, each region is also a permutation of 1 through 9. The less the number of filled cells (also called *entries* or *hints*) in a Sudoku puzzle, the more difficult the puzzle (for human). Gordon Royle has a collection of 47,386 distinct Sudoku configurations with exact 17 filled cells [Roy07]. It is an open question whether 17 is the minimum number of entries for a standard Sudoku puzzle to have a unique solution. It is also an open question whether Royle’s collection of 47,386 puzzles is complete (up to isomorphism).

Results obtained by model generators

Since the standard Sudoku puzzle is an easy problem for SAT solvers, several people created SAT-based Sudoku solvers [Web05, Hal07, LO06].

For completing a general partial quasigroup problem, Gomes and Shmoys suggested it as a structured benchmark domain for the study of constraint satisfaction (CSP) and SAT methods [GS02]. The quasigroup completion problem provides a good alternative to other benchmarks because it has structure not found in other randomly generated SAT problems and it can be set to generate only satisfiable instances (e.g., good for testing local search methods). Moreover, the underlying structure of Latin squares has many similarity comparing to many real-world applications. In [GS02], Gomes and Shmoys compared three approaches to this problem: (1) the CSP approach; (2) a hybrid linear programming/CSP approach and (3) the SAT-based approach. It has been shown that none of these methods uniformly dominates the others on this domain and the SAT-based approach can solve some of the hardest problem instances, providing the number of SAT clauses remains manageable.

To complete a partial quasigroup using a SAT solver, we generate a positive unit clause for each filled cell and then use the clauses (21.1)-(21.6) presented in the beginning of this section. This is called “extended encoding” in [GS02]. If

the clauses 21.3, 21.5 and 21.6 are dropped, it is called “minimal encoding”. It is reported that the extra clauses in the extended encoding increase propagation dramatically and allow them to solve much larger problems considerably faster. Using the extended encoding, they could complete Latin squares of size up to 60, using SAT solvers.

21.2.1.2. Orthogonality of Latin Squares

People are interested in Latin squares that satisfy a set of constraints. For instance, the Sudoku puzzle requires that a Latin square also satisfies the constraint that each one of the nine regions is a permutation of 1 through 9. These constraints are often expressed in terms of the operator $*$ plus some free (or universally quantified) variables. For example, idempotent Latin squares are those that satisfy $x * x = x$. In this section, all the Latin squares are assumed to be idempotent, unless stated otherwise. Some constraints involve more than one Latin square.

The first constraint we consider here, and perhaps the most important constraint in the study of Latin squares, is the orthogonality of two Latin squares. Two $n \times n$ matrices over the domain S ($S, *$) and (S, \star) are said to be *orthogonal* if for any two elements s, t of S , the set $\{(x, y) \mid x * y = s, x \star y = t\}$ is singleton. Intuitively, if both matrices are viewed as two chessboards and are placed one over the other, the pair of values at any position (i.e., the pair (s, t) at position (x, y)) is unique. Note that the definition of orthogonality does not require that the two matrices being Latin squares; when the two orthogonal matrices are Latin squares, we say they are a pair of orthogonal Latin squares.

The orthogonality of two matrices of the same order can be specified as the following constraint: For all elements x, y, z, w of S

$$(x * y = z * w) \wedge (x \star y = z \star w) \Rightarrow (x = z \wedge y = w) \quad (21.7)$$

One problem of great interests is to prove the existence of a set of mutually orthogonal Latin squares (MOLS) of certain size. The following remark is quoted from [Ce96]:

Euler’s 36 Officers Problem. A very curious question, which has exercised for some time the ingenuity of many people, has involved me in the following studies, which seem to open a new field of analysis, in particular the study of combinations. The question revolves around arranging 36 officers to be drawn from 6 different ranks and also from 6 different regiments so that they are arranged in a square so that in each line (both horizontal and vertical) there are 6 officers of different ranks and different regiments.

If the ranks and regiments of these 36 officers arranged in a square are represented, respectively, by two 6×6 Latin squares, Euler’s 36 officers problem asks if there exist two MOLS of size 6. Euler went on to conjecture that such an $n \times n$ square does not exist for $n = 6$, nor does one exist whenever $n \equiv 2 \pmod{4}$. This was known as the *Euler conjecture* until its disproof in 1959. That year, Parker, Bose and Shrikhande were able to construct a pair of orthogonal Latin squares of order 10, and provided a construction for the remaining even values of n that are not divisible by 4 (of course, excepting $n = 2$ and $n = 6$) [BS60]. Today’s model

generators can find a pair of such Latin squares in no time. However, it remains a great challenge to find a set of three mutually orthogonal Latin squares of order 10.

Let $N(n)$ be the maximum number of Latin squares in a set of MOLS of size n .

Theorem 21.2.2. [Ce96]

- (a) For every $n > 1$, $1 \leq N(n) \leq n - 1$.
- (b) If $q = p^e$, where p is a prime, then $N(q) = q - 1$.
- (c) If n is sufficiently large, $N(n) \geq n^{\frac{1}{14.8}}$.
- (d) For small n , which is not a prime power, the following results are known:

n	6	10	12	14	15	18	20	21	22	24
$N(n)$	1	≥ 2	≥ 5	≥ 3	≥ 4	≥ 3	≥ 4	≥ 5	≥ 3	≥ 5

The above table shows that there are many open problems for $N(n)$. For instance, it is still unknown if there exists a set of three pairwise-orthogonal Latin squares of size 10. A positive answer to this problem will have a tremendous effect in combinatorial theory. In 1989, Lam reported that no finite projective plane of order 10 exists [Lam91]. This result implies that $N(10) < 9$, i.e., there are no nine MOLS of size 10. This example shows that Latin squares have a rich relationship with other objects in combinatorics.

Given three mutually orthogonal Latin squares of order n , say A , B , and C , if $A = B^T$ and $C = C^T$, where X^T is the transpose of X , we say (A, B, C) is an SOLSSOM(n) (Self-Orthogonal Latin Square with a Symmetric Orthogonal Mate).

Theorem 21.2.3. [ABZZ00] An SOLSSOM(n) exists if and only if $n \geq 4$, except for $n = 6$ and possibly for $n = 10, 14$.

Results obtained by model generators

The orthogonality of two Latin squares defined by formula 21.7 can be easily converted into two clauses:

$$x * y = s \wedge z * w = s \wedge x * y = t \wedge z * w = t \Rightarrow x = z \quad (21.8)$$

$$x * y = s \wedge z * w = s \wedge x * y = t \wedge z * w = t \Rightarrow y = w \quad (21.9)$$

In [Zha97] as well as in section 21.3.1, how to encode the orthogonality constraint is discussed.

The author of the present chapter has spent a huge amount of effort to decide if $N(10) \geq 3$. Since 1997, at least 10 linux workstations have been used day and night to crack this problem. Except that we confirmed the nonexistence of three MOLS of size 10 satisfying some additional constraints, we have not succeeded in this problem. Despite the fact that we spent 10 years, sometimes using a cluster of 40 linux machines, only a small portion of the whole search space has been explored. Because of the randomization of the search method [Zha02], we believe that the maximum number of MOLS of size 10 is most likely 2, that is, $N(10) = 2$.

For Theorem 21.2.3 regarding the existence of SOLSSOM(n), the model generator solved indirectly two previously-open cases: $n = 66, 70$ [ABZZ00].

21.2.1.3. Conjugates of Latin Squares

Given a Latin square $(S, *)$ and values of any two variables in $x * y = z$, we can uniquely determine the value of the third variable. We may therefore associate with $(S, *)$ an operation \star such that $x \star z = y$ if and only if $x * y = z$. It is easy to see that (S, \star) is also a Latin square. (S, \star) is one of the six *conjugates* of $(S, *)$. These are defined via the six operations $*_{ijk}$, where $\{i, j, k\} = \{1, 2, 3\}$:

$$(x_i *_{ijk} x_j = x_k) \iff (x_1 * x_2 = x_3)$$

We shall refer to $(S, *_{ijk})$ as the (i, j, k) -conjugate of $(S, *)$. Whenever S is understood to be common, we will simply refer to $*_{ijk}$ as the (i, j, k) -conjugate of $*$. It is easy to see that the $(1, 2, 3)$ -conjugate is the same as the original Latin square, and the $(2, 1, 3)$ -conjugate is the transpose of the original Latin square.

Example 21.2.1. Here are the six conjugates of a small Latin square:

(a)	(b)	(c)	(d)	(e)	(f)
1 4 2 3	1 2 4 3	1 3 2 4	1 2 4 3	1 3 4 2	1 3 2 4
2 3 1 4	4 3 1 2	2 4 1 3	3 4 2 1	3 1 2 4	3 1 4 2
4 1 3 2	2 1 3 4	4 2 3 1	2 1 3 4	2 4 3 1	4 2 3 1
3 2 4 1	3 4 2 1	3 1 4 2	4 3 1 2	4 2 1 3	2 4 1 3

(a) a Latin square; (b) its $(2, 1, 3)$ -conjugate; (c) its $(3, 2, 1)$ -conjugate; (d) its $(2, 3, 1)$ -conjugate; (e) its $(1, 3, 2)$ -conjugate; (f) its $(3, 1, 2)$ -conjugate. \square

In this subsection, let us focus on orthogonality of conjugate Latin squares. If we replace \star in formula 21.7 by $*_{213}$, $*_{321}$, and $*_{312}$, respectively, we have the following constraints:⁴

$$\text{QG0: } (x * y = z * w \wedge x *_{213} y = z *_{213} w) \Rightarrow (x = z \wedge y = w)$$

$$\text{QG1: } (x * y = z * w \wedge x *_{321} y = z *_{321} w) \Rightarrow (x = z \wedge y = w)$$

$$\text{QG2: } (x * y = z * w \wedge x *_{312} y = z *_{312} w) \Rightarrow (x = z \wedge y = w)$$

These constraints can be rephrased uniformly in $*$ as:

$$\text{QG0: } (x * y = z * w \wedge y * x = w * z) \Rightarrow (x = z \wedge y = w)$$

$$\text{QG1: } (x * y = z * w \wedge u * y = x \wedge u * w = z) \Rightarrow (x = z \wedge y = w)$$

$$\text{QG2: } (x * y = z * w \wedge y * u = x \wedge w * u = z) \Rightarrow (x = z \wedge y = w)$$

That is, a Latin square satisfying QG0, QG1 or QG2, will be orthogonal to its $(2, 1, 3)$ -, $(3, 2, 1)$ -, or $(3, 1, 2)$ -conjugates, respectively. For example, the Latin square (a) in Example 21.2.1 satisfies QG2, since it is orthogonal to (f), its $(3, 1, 2)$ -conjugate. A Latin square satisfying QG0 is said to be *self-orthogonal*. It is easy to see that the first Latin square of an SOLSSOM(n) is self-orthogonal.

We say that constraint C' is a *conjugate-implicant* of constraint C if whenever a Latin square satisfies C , one of its conjugates satisfies C' . We say two constraints are *conjugate-equivalent* if they are conjugate-implicants of each other.

⁴We will continue to use the nomenclature QG0–QG15 introduced in [FSB93, SZ94, Zha97]. These meaningless code names may facilitate the communication among the users of model generators who have little knowledge of Latin squares. SATO uses these code names to generate propositional clauses. For instance, the command `sato -Q2 -G10 -o3` will output the propositional clauses for Latin squares of order 10 satisfying QG2.

Table 21.1. Conjugate-Orthogonality Constraints

	*	*132	*213	*231	*312	*321
*	—	QG1'	QG0	QG2'	QG2	QG1
*132	QG1'	—	QG2	QG1''	QG0''	QG2''
*213	QG0	QG2	—	QG1	QG1'	QG2'
*231	QG2'	QG1''	QG1	—	QG2''	QG0'
*312	QG2	QG0''	QG1'	QG2''	—	QG1''
*321	QG1	QG2''	QG2'	QG0'	QG1''	—

The orthogonality of a pair of conjugates can be logically equivalent to, or conjugate-equivalent to, orthogonality of other pairs of conjugates. These relationships are summarized in Table 21.1 (taken from [SZ94]). Each table entry is a code name for a constraint that is defined to be logically equivalent to the orthogonality of its row and column labels. For example, QG1 is defined by orthogonality of * and *₃₂₁, but could have been defined equivalently by orthogonality of *₂₁₃ and *₂₃₁. The constraints QG0, QG0' and QG0'' are conjugate-equivalent; so are QG1, QG1' and QG1'', and so are QG2, QG2' and QG2''.

We will use $QG_i(v)$ to denote an idempotent Latin square of order v satisfying constraint QG_i .

Theorem 21.2.4. [Ce96, Zha97]

- There exists a $QG_0(n)$ for $n \geq 1$, except $n = 2, 3, 6$.
- There exists a $QG_1(n)$ for $n \geq 1$, except $n = 2, 3, 6$.
- There exists a $QG_2(n)$ for $n \geq 1$, except $n = 2, 3, 4, 6, 10$.

Results obtained by model generators

Despite of all the developments of new model generators, QG0, QG1 and QG2 remain to be difficult model generation problems. It is noteworthy to point out that model generators removed five previously-open cases in Theorem 21.2.4. That is, using some incomplete methods, the existence of QG2(12) was first established by Stickel's DDPP [SFS95], and the existences of QG1(12), QG2(14) and QG2(15) were first established by SATO [ZBH96]. The nonexistence of QG2(10) was first established by Olivier Dubois and Gilles Dequen using a special-purpose program [DD01] where the constraints are treated in the form of CSP. This result was confirmed subsequently by SATO. Note that the accumulated time spent by SATO on QG2(10) has been over one year on three linux workstations and the search space has finally been exhausted.

21.2.1.4. Holey Latin Squares

Among the Latin squares satisfying certain constraints, people are often interested in those squares with holes, i.e., some subsquares of the square are missing. Holey Latin squares are special cases of partial Latin squares. The existence of these holey Latin squares is very useful in the construction of Latin squares of large orders.

Formally, let S be a set and $\mathcal{H} = \{S_1, \dots, S_m\}$ be a set of subsets of S . A *holey Latin square* having *hole set* \mathcal{H} , denoted by $(S, \mathcal{H}, *)$, is a $|S| \times |S|$ matrix L , indexed by S , satisfying the following properties:

1. every cell of L either contains an element of S or is empty,
2. every element of S occurs at most once in any row or column of L ,
3. the sub-matrices indexed by $S_i \times S_i$ are empty for $1 \leq i \leq m$ (these sub-matrices are referred to as *holes*),
4. element $s \in S$ occurs in row or column t if and only if $(s, t) \notin S_i \times S_i$ for any $1 \leq i \leq m$.

A holey Latin square is a partial Latin square but the opposite may not be true because of Constraints 3 and 4. Two types of holey Latin squares are of particular interests: (a) $\mathcal{H} = \{S_1\}$ and (b) $\mathcal{H} = \{S_1, \dots, S_m\}$ is a partition of S .

Holey Latin squares in case (a) are called *incomplete Latin squares*. Given a Latin square constraint QGi , an incomplete idempotent Latin square indexed by S with a single hole S_1 is often denoted by $IQGi(|S|, |S_1|)$. We may consider an incomplete Latin square as one indexed by S with a subsquare indexed by S_1 “missing”; if we fill a Latin square indexed by S_1 into the hole, the result should be a Latin square of order $|S|$. Without loss of generality, the missing subsquare can be assumed to be in the bottom right corner of the Latin square.

Holey Latin squares in case (b) are called *frame Latin squares*. If $|S_1| = \dots = |S_m|$, any frame Latin square satisfying QGi is often denoted by $FQGi(h^m)$, where $h = |S_1| = \dots = |S_m|$ and $|S| = hm$. Note that every idempotent $QGi(n)$ can be considered as an $FQGi(1^n)$.

A necessary condition for the existence of incomplete Latin squares satisfying $IQGi(n, k)$, $i = 0, 1, 2$, is $n \geq 3k + 1$. The known results regarding the existences of incomplete idempotent $IQGi(n, k)$, $i = 0, 1, 2$, are summarized in the following theorem:

Theorem 21.2.5. [Ce96, Zha97]

(a) For every integer $n \geq 1$, an $IQG0(n, k)$ exists if and only if $n \geq 3k + 1$, except $(n, k) = (6, 1), (8, 2)$, and except possibly $(n, k) = (20, 6)$.

(b) For every integer $n \geq 1$, an $IQG1(n, k)$ exists if $n \geq (13/4)k + 88$. For $2 \leq k \leq 6$, an $IQG1(n, k)$ exists if and only if $n \geq 3k + 1$ except possibly $(n, k) = (11, 3)$.

(c) For any integer $n \geq 1$, an $IQG2(n, k)$ exists if $n \geq (10/3)k + 68$. For $2 \leq k \leq 5$, an $IQG2(n, k)$ exists if and only if $n \geq 3k + 1$ except $(n, k) = (8, 2)$ and except possibly $k = 4$ and $n \in \{35, 38\}$.

The existence of frame $FQGi(h^n)$, $i = 0, 1, 2$, have almost completely decided.

Theorem 21.2.6. [Ce96, Zha97]

(a) For any $h \geq 2$, there exists an $FQG0(h^n)$ if and only if $n \geq 4$.

(b) For any $h \geq 2$, there exists an $FQG1(h^n)$ if and only if $n \geq 4$, except possibly $(h, n) = (13, 6)$.

(c) For any $h \geq 2$, there exists an $FQG2(h^n)$ if and only if $n \geq 4$, except possibly $(h, n) = (2t + 1, 4)$ for $t \geq 1$.

Results obtained by model generators

Many cases in Theorem 21.2.5 were first solved by a model generator. For instance, 16 cases of IQG0(n, k), including $(n, k) = (14, 4)$, was found by SATO. The last two cases, $(26, 8)$ and $(32, 10)$, were solved by SATO after the publication of the first edition of this Handbook. It is reported in [ZB95] that the existence of IQG1(n, k) was established by SATO for $(n, k) = (30, 5)$, $k = 2$ and $n \in \{16, 17, 19, 20, 21, 23\}$ and $k = 3$ and $n \in \{20, 21, 24, 25, 26, 28, 29, 30\}$. For IQG2(n, k), the nonexistence of IQG2(8, 2) was first established by FINDER [SFS95]. SATO solved more than forty previously-open cases [BZ98].

Again, model generators made a contribution in Theorem 21.2.6. It is reported in [ZB95] and [BZ98] that more than thirty cases in this theorem were solved by SATO.

In [Zha97], an easy way of encoding holey quasigroups in SAT is presented which involves only adding some negative units clauses in the encoding of normal quasigroups.

21.2.1.5. Short Conjugate-Orthogonal Identities

Many Latin square constraints are in the form of equations, called *quasigroup identities* [BZ92]. For instance, QG5, i.e., $((y * x) * y) * y = x$, is one of the first constraints studied by many model generators. A Latin square identity is said to be *nontrivial* if it is not a tautology and it is consistent with the specification of a Latin square. In this subsection as well as in the next subsections, we will focus on several nontrivial quasigroup identities.

A quasigroup identity is called a *short conjugate-orthogonal identity* if it is of the form $(x *_{ijk} y) * (x *_{abc} y) = x$, where $*_{ijk}, *_{abc} \in \{*, *_{213}, *_{132}, *_{312}, *_{231}, *_{321}\}$ [Eva75].

Theorem 21.2.7. [Eva75, Ben87] *Every nontrivial short conjugate-orthogonal identity is conjugate-equivalent to one of the following:*

Code Name	Identity
QG3	$(x * y) * (y * x) = x$
QG4	$(y * x) * (x * y) = x$
QG5	$((y * x) * y) * y = x$
QG6	$(x * y) * y = x * (x * y)$
QG7	$(y * x) * y = x * (y * x)$
QG8	$x * (x * y) = y * x$
QG9	$((x * y) * y) * y = x$

Many short conjugate-orthogonal identities possess interesting properties [BZ92]. Given a Latin square identity, people are interested in whether some of the three constraints, QG0, QG1 and QG2, are its conjugate-implicants. For instance, it is known that QG2 is a conjugate-implicant of QG4; if you are interested in a QG2(12), you may look for a QG4(12). Indeed, that is how QG2(12) was found by Mark Stickel [SFS95].

Table 21.2. Conjugate-Orthogonality Implications for QG3–QG9

Identity	Conjugate-Implicants
QG3	QG0
QG4	QG0, QG2
QG5	QG1, QG2
QG6	QG1
QG7	QG1, QG2
QG8	QG0, QG1
QG9	QG1

Evans [Eva75] shows that if a Latin square satisfies the short conjugate-orthogonal identity $(x *_{ijk} y) * (x *_{abc} y) = x$, then its (i, j, k) -conjugate is orthogonal to its (a, b, c) -conjugate. Table 21.2 (taken from [SZ94]) shows which of QG0–QG2 are conjugate-implicants of QG3–QG9 [SZ94].

The existences of idempotent $QGi(n)$, $3 \leq i \leq 9$, are summarized in the following theorem:

Theorem 21.2.8. [Ce96, Zha97]

- (a) *There exists a $QG3(n)$ if and only if $n \equiv 0$ or $1 \pmod{4}$.*
- (b) *There exists a $QG4(n)$ if and only if $n \equiv 0$ or $1 \pmod{4}$, except $n = 4$ and 8 .*
- (c) *There exists a $QG5(n)$ for $n \geq 1$, except $n \in \{2, 3, 4, 6, 9, 10, 12, 13, 14, 15, 16\}$ and except possibly $n \in \{18, 20, 22, 24, 26, 28, 30, 34, 38, 39, 42, 44, 46, 51, 52, 58, 60, 62, 66, 68, 70, 72, 74, 75, 76, 86, 87, 90, 949, 96, 98, 99, 100, 102, 106, 108, 110, 114, 116, 118, 122, 132, 142, 146, 154, 158, 164, 174\}$.*
- (d) *There exists a $QG6(n)$ for $n \equiv 0$ or $1 \pmod{4}$, except $n = 5, 12$ and except possibly $n \in \{20, 21, 24, 41, 44, 48, 53, 60, 69, 77, 93, 96, 101, 161, 164, 173\}$; for $n \equiv 2$ or $3 \pmod{4}$, there does not exist a $QG6(n)$ for $n \leq 15$.*
- (e) *There exists a $QG7(n)$ for all $n \equiv 1 \pmod{4}$ except possibly $n = 33$; for $n \not\equiv 1 \pmod{4}$, there does not exist a $QG7(n)$ for $n \leq 16$.*
- (f) *There exists a $QG8(n)$ for all $n \geq 1$, except $n \in \{2, 3, 6, 7, 8, 10, 12, 14, 15, 18\}$, and except possibly $n \in \{22, 23, 26, 27, 30, 34, 38, 42, 43, 46, 50, 54, 62, 66, 74, 78, 90, 98, 102, 114, 126\}$.*
- (g) *There exists a $QG9(n)$ if and only if $n \equiv 1 \pmod{3}$.*

Theorem 21.2.9. [Ce96, Zha97]

- (a) *For any $n \geq 7$, an $IQG3(n, 2)$ exists if and only if $n \equiv 2$ or $3 \pmod{4}$ except $n = 10$.*
- (b) *For $h \geq 2$, an $FQG3(h^n)$ exists if and only if $hn(n-1) \equiv 0 \pmod{4}$ with exceptions of $(h, n) \in \{(1, 5), (1, 9), (2, 4)\}$ and possible exception of $(h, n) = (6, 4)$.*
- (c) *For any $n \geq 7$, an $IQG4(n, 2)$ exists if and only if $n \equiv 2$ or $3 \pmod{4}$ except possibly $n = 19, 23, 27$; for any $n \geq 10$ and $k = 3$ or 7 , an $IQG4(n, k)$ exists if and only if $n \equiv 2$ or $3 \pmod{4}$.*
- (d) *For $h \geq 2$, an $FQG4(h^n)$ exists if and only if $h^2n(n-1) \equiv 0 \pmod{4}$ with exception of $(h, n) = 2^4$ and $(2k+1)^4$ for $k \geq 1$.*

In [MZ07], Ma and Zhang studied the existence of *large sets* satisfying the short quasigroup identities. Two idempotent quasigroups $(S, *)$ and (S, \star) are said to be *disjoint* if for any $x, y \in S$, $x * y \neq x \star y$ whenever $x \neq y$. A set of $n - 2$ idempotent quasigroups over S , where $|S| = n$, is said to be a *large set* if any two quasigroups in the set are disjoint. For i from 3 to 9, let $LSQG_i(n)$ denote a large set of order n such that each quasigroup in the large set satisfies QG_i .

Theorem 21.2.10. [Ce96, MZ07]

- (a) *There exist $LSQG_i(n)$ for $(i, n) \in \{(3, 8), (6, 8), (8, 4), (9, 4)\}$;*
- (b) *There exist no $LSQG_i(n)$ for $(i, n) \in \{(5, 7), (5, 8), (8, 4), (9, 2), (9, 4), (9, 6), (9, 7), (11, 4), (11, 5)\}$.*

Results obtained by model generators

Model generators have been used to establish several cases in Theorem 21.2.8: $QG_3(12)$ was first found by FINDER [SFS95]; $QG_4(12)$ was first found by DDPP [SFS95].

All the new results regarding QG_5 were negative: the nonexistence of $QG_5(9)$ was first established by FALCON [Zha96]; the nonexistence of $QG_5(10)$ (without idempotency) and $QG_5(12)$ was by MGTP [SFS95]; the nonexistence of $QG_5(13)$, $QG_5(14)$ and $QG_5(15)$ was by DDPP [SFS95]; the nonexistence of $QG_5(14)$ (without idempotency) and $QG_5(16)$ was by SATO [ZBH96].

$QG_6(9)$ was first found by MGTP [SFS95] and $QG_6(17)$ was by MACE [McC94]. The nonexistence of $QG_6(12)$ was first established by MGTP [SFS95]. For the orders not congruent to 0 or 1 (mod 4), the nonexistence of $QG_6(10)$, $QG_6(11)$ and $QG_6(14)$ was reported in [SFS95] and that of $QG_6(15)$ was by SATO [ZBH96]; the other cases remain open.

All the new results regarding QG_7 were negative: the nonexistence of $QG_7(i)$ was first established by MGTP for $i = 7, 8, 10$ [SFS95]; by FINDER for $i = 11, 14, 15$ [SFS95]; by DDPP for $i = 12$ and 16 [SFS95].

The nonexistence of $QG_8(15)$ was first established by DDPP [SFS95].

Model generators are also used in the establishment of Theorem 21.2.9. In particular, Slaney, Fujita and Stichel reported the following results in [SFS95]: the existence of $IQG_3(11, 3)$, $IQG_4(10, 2)$ and $IQG_4(11, 2)$; the nonexistence of $IQG_3(10, 2)$. Later, Stichel found an $IQG_3(14, 2)$. SATO found an $IQG_4(14, 2)$ and an $IQG_4(15, 2)$ [ZBH96].

Holey Latin squares satisfying QG_5 – QG_9 have not been fully investigated. Slaney, Fujita and Stichel reported the following nonexistence results in [SFS95]: $IQG_5(n, k)$ for $(n, k) = (7, 2), (9, 2), (11, 2), (14, 3)$ and $(16, 6)$, and $IQG_6(n, k)$ for $2 \leq k < n \leq 11$.

The results in Theorem 21.2.10 are obtained with the help of SEM [ZZ95] for generating subproblems with additional constraints of symmetry breaking [HMG⁺18]. By exhaustive search, they found that the maximum set size of mutually disjoint $QG_i(n)$ quasigroups is 6 for $(i, n) = (9, 2), (9, 4), (9, 6)$, and $(9, 7)$; 4 for $(i, n) = (4, 11)$; and 2 for $(i, n) = (5, 11)$, respectively [HMG⁺18]. There are remaining open cases for small orders [MZ07, HMG⁺18].

In [SFS95, Zha97], the issue of how to encode short quasigroup identities into SAT is discussed and it is shown that adding extra constraints helps the search

when the size of problems increases.

21.2.2. Ramsey Numbers

Given two positive integers r and s , Ramsey's Theorem [Ram30] of graph theory states that a sufficiently-large simple graph either contains a clique of size r (there exists an edge between any pair of these vertices) or an independent set of size s (no edges among these vertices). That is, there exists an integer $R(r, s)$ such that for every graph of size at least $R(r, s)$, it contains either a clique of size r , or an independent set of size s . Ramsey first proved this result in 1930 which initiated the combinatorial theory, now called Ramsey theory [GRS90], that seeks regularity amid disorder: general conditions for the existence of substructures with regular properties. In this example, the regular properties are either cliques or independent sets, that is, subsets of vertices either all connected or with no edges at all.

The least number $R(r, s)$ in Ramsey's theorem is known as *Ramsey number*. An upper bound for $R(r, s)$ can be extracted from Ramsey's proof of the theorem, and other arguments give lower bounds. Since there is a vast gap between the tightest lower bounds and the tightest upper bounds, there are very few numbers r and s for which we know the exact value of $R(r, s)$. The following table shows the latest information regarding these numbers [Rad05].

$r \setminus s$	2	3	4	5	6	7	8
2	2						
3	3	6					
4	4	9	18	76			
5	5	14	25	43 – 49			
6	6	18	35 – 41	59 – 87	102 – 165		
7	7	23	49 – 61	80 – 143	113 – 298	205 – 540	
8	8	28	56 – 84	101 – 216	127 – 495	216 – 1031	282 – 1870

For instance, we do not know the exact value of $R(5, 5)$, although we know that it lies between 43 and 49. Computing a lower bound n for $R(r, s)$ usually requires finding a simple graph of $n - 1$ vertices which has neither an r -clique nor an s -independent set. For instance, if we could find a graph of 43 vertices which has neither a clique of size 5 nor an independent set of size 5, we would increase the lower bound of $R(5, 5)$ to 44. Modern SAT solvers may be used for this task. In fact, we define a set $LR(r, s, n)$ of propositional clauses such that $LR(r, s, l)$ are satisfiable if and only if there exists a graph of l vertices which has neither clique of size r and nor independent set of size s . Then $LR(r, s, l)$ is fed to a SAT solver which decides its satisfiability.

Let $V = \{1, 2, \dots, n\}$ be the set of vertices of a simple graph. Let $p_{i,j}$ be the propositional variable which is true if and only if edge (i, j) is in the graph, where $1 \leq i < j \leq n$. Then for any subset $S \subset V$, the subgraph induced by S is not a clique if the following clause is true:

$$\text{not-a-clique}(S): \bigvee_{i,j \in S} \neg p_{i,j}$$

Note that the above clause has $m(m-1)/2$ literals, where $m = |S|$. Similarly, the subgraph induced by S is not an independent set if the following clause is true:

$$\text{not-an-independent-set}(S): \bigvee_{i,j \in S} p_{i,j}$$

Using $\text{not-a-clique}(S)$ and $\text{not-an-independent-set}(S)$, $LR(r, s, n)$ can be defined as follows:

$$\text{not-a-clique}(X) \text{ for every subset } X \in V, |X| = r; \quad (21.10)$$

$$\text{not-an-independent-set}(Y) \text{ for every subset } Y \in V, |Y| = s. \quad (21.11)$$

Some earlier SAT solvers use $LR(r, s, n)$ as benchmark problems.⁵ While a complete SAT solver can check lower bounds for small r and s , it becomes intractable for large values. For instance, $LR(5, 5, 43)$ contains more than 1.7 millions clauses. In general, the number of clauses in $LR(r, s, n)$ is $\frac{1}{2}r(r-1)\binom{r}{n} + \frac{1}{2}s(s-1)\binom{s}{n}$, where $\binom{m}{n} = n!/(m!(n-m)!)$. While the SAT approach has not yet improved any lower bound in the above table, it is worth pointing out that the SAT approach is successful in solving another problem of the Ramsey theory, i.e., the Van der Waerden numbers.

21.2.3. Van der Waerden Numbers

Given a positive integer m , let $[m]$ denote the set $\{1, \dots, m\}$. A *partition* of a set X is a collection $\mathcal{A} = \{A_1, \dots, A_k\}$ of mutually disjoint subsets of X such that $A_1 \cup \dots \cup A_k = X$. Elements of \mathcal{A} are commonly called *blocks*. An *arithmetic progression* of length n is a sequence of n integers a_1, \dots, a_n such that $a_i = a_1 + (i-1)b$ for all $1 \leq i \leq n$ and some integer b .

The Van der Waerden theorem [DMT04] says that, for sufficiently large m , if $[m]$ is partitioned into a few blocks, then one of these blocks has to contain an arithmetic progression of a desired length.

Theorem 21.2.11. (Van der Waerden theorem) *For any positive integers k and l , there exists m such that for every partition $\langle A_1, \dots, A_k \rangle$ of $[m]$, there exists a block A_i which contains an arithmetic progression of length at least l .*

The *Van der Waerden number* $W(k, l)$ is defined to be the least number m for which the assertion in the above theorem holds. For $l = 2$, $W(k, 2) = k + 1$. For $l > 2$, little is known about $W(k, l)$ as no closed formulas have been found so far. Only five exact values are known:

$k \setminus l$	3	4	5
2	9	35	178
3	27	?	?
4	76	?	?

⁵Earlier versions of SATO can be used to produce $LR(r, s, n)$ as follows: `sato -Rn -Pr -Qs -o3`.

The following general results regarding the lower bounds of $W(w, l)$ are known: Erdős and Rado [ER52] provided a non-constructive proof for the inequality

$$W(k, l) > (2(l-1)k^{l-1})^{1/2}.$$

When $k = 2$ and $l - 1$ is a prime number, Berlekamp [Ber68] showed that

$$W(k, l) > (l-1)2^{l-1}.$$

This implies that $W(2, 6) > 160$ and $W(2, 8) > 896$.

Using SAT solvers, Dransfield, Marek and Truszczynski [DMT04] found larger lower bounds for several small Van der Waerden numbers and improved significantly the results implied by the general results of Erdős and Rado [ER52] and Berlekamp [Ber68]. Also using SAT solvers, Kouril and Franco reported $W(2, 6) \geq 1132$ in [KF05] and claimed $W(2, 6) = 1132$ after 253 days of computational time. Herwig, Heule, Van Lambalgen and Van Maaren [HHvLvM07] reported exciting results using a new approach called the *cyclic zipper method*.

Following the notation used in [DMT04], let $\text{vdW}_{k,l,m}$ denote a set of propositional clauses such that $\text{vdW}_{k,l,m}$ is satisfiable if and only if there exists a partition of $[w]$ into k blocks and none of the blocks contains an arithmetic progression of length l . Such a partition is called a *Van der Waerden certificate* of $W(k, l) > m$. That is, if $\text{vdW}_{k,l,m}$ is satisfiable, then $W(k, l) > m$. The following mk propositional variables are used in $\text{vdW}_{k,l,m}$: $\text{in}(i, b)$ for each $i \in [m]$ and each $b \in [k]$. Intuitively, $\text{in}(i, b)$ is true if and only if integer i is in block b .

The set $\text{vdW}_{k,l,m}$ contains the following three types of propositional clauses: For every $i, d \in [m]$, $b, b' \in [k]$, where $b < b'$ and $i + (l-1)d \leq m$,

$$\begin{aligned} (\text{vdW1}) \quad & \neg \text{in}(i, b) \vee \neg \text{in}(i, b') \\ (\text{vdW2}) \quad & \text{in}(i, 1) \vee \dots \vee \text{in}(i, k) \\ (\text{vdW3}) \quad & \neg \text{in}(i, b) \vee \neg \text{in}(i + d, b) \vee \dots \vee \neg \text{in}(i + (l-1)d, b) \end{aligned}$$

Intuitively, vdW1 and vdW2 together specify that each integer of $[m]$ is in exactly one of the k blocks. The last clause, vdW3, says that no block contains an arithmetic progression of length l .

Proposition 21.2.1. [DMT04] For positive integers k, l and m , with $l \geq 2$, $m < W(k, l)$ if and only if $\text{vdW}_{k,l,m}$ is satisfiable.

Results obtained by model generators

Using complete SAT solvers, Dransfield et al. could verify that $W(4, 3) = 76$ by proving that $\text{vdW}_{4,3,75}$ is satisfiable and $\text{vdW}_{4,3,76}$ is unsatisfiable. However, complete SAT solvers become ineffective as the size grows. Dransfield et al. estimated that the total size of clauses (vdW1) is $\Theta(mk^2)$; the size of (vdW2) is $\Theta(mk)$ and the size of (vdW3) is $\Theta(m^2)$. So the total size of $\text{vdW}_{k,l,m}$ is $\Theta(mk^2 + m^2)$. Using an incomplete SAT solver, Dransfield et al. believed that they found nine new lower bounds of $W(k, l)$, including $W(5, 3) > 125$ and $W(2, 6) > 341$, as they were unaware of the work of Rabung [Rab79].

Dransfield et al. used the well-known incomplete SAT solver WSAT. Their way of using WSAT is quite interesting. Let $m_1 < \dots < m_s = m$ be a sequence of

positive integers. To find a model of $\text{vdW}_{k,l,m}$, they look for models of vdW_{k,l,m_1} , ..., vdW_{k,l,m_s} , in consecutive order. The algorithm proceeds as follows.

1. Let $i = 1$, generate a random interpretation of vdW_{k,l,m_1} .
2. Call $\text{wsat}(\text{vdW}_{k,l,m_i})$ until it stops.
3. When a model of vdW_{k,l,m_i} is found in step 2, if $i = s$, stop and output this model; otherwise, use the model as the initial interpretation of $\text{vdW}_{k,l,m_{i+1}}$, let $i = i + 1$ and go to step 2.
4. If $\text{wsat}(\text{vdW}_{k,l,m_i})$ fails to find a model at step 2 and the time allows, let $i = 1$ and goto step 1.

Dransfield et al.'s work caught the attention of the SAT community. In [KF05], Kouril and Franco reported that $W(2, 6) > 1131$. Using the cyclic zipper method, Herwig et al [HHvLvM07] found ten new lower bounds of $W(k, l)$. Their approach stems from the SAT approach by first visualizing the Van der Waerden certificates of small orders and observing their regularities such as symmetry and repetition. They then look for special certificates of large sizes which possess these regularities using SAT solvers. These latest results are listed in the following table, where $W(5, 3) > 125$ was established in [DMT04], $W(2, 6) > 1131$ was done in [KF05]. Ten entries were established in [HHvLvM07]. While this chapter is being written, four new lower bounds are found: $W(5, 3) > 170$, $W(6, 5) > W(5, 5) > 98740$, and $W(5, 6) > 540025$.⁶ The case of $W(5, 3) > 170$ is particularly interesting as it was found by a SAT solver.

$k \setminus l$	3	4	5	6	7	8	9
2	9	35	178	> 1131	> 3703	> 11495	> 41265
3	27	> 292	> 2173	> 11191	> 43855	> 238400	
4	76	> 1048	> 17705	> 91331	> 393469		
5	> 170	> 2254	> 98740	> 540025			
6	> 207	> 9778	> 98740	> 633981			

Given the success of the various SAT approaches on the Van der Waerden numbers, it can be expected that the same strategy may work for other problems of the Ramsey theory, including the Ramsey numbers in the previous subsection.

21.2.4. Covering Arrays

A *covering array*, $\text{CA}(n; t, k, v)$, of size n , strength t , degree k , and order v is an $n \times k$ matrix (n rows and k columns) A on v symbols with the property that the projection of each row on any t columns contains all possible t -tuples of v symbols (v^t possibilities). That is, suppose A is on a set V of v symbols, for any t -tuples $\vec{x} \in V^t$ and any t columns $1 \leq c_1 < c_2 < \dots < c_t \leq k$, there exists a row i such that $\vec{x} = A[i, c_1]A[i, c_2] \cdots A[i, c_t]$.

Figure 21.2.4 displays a $\text{CA}(8; 3, 4, 2)$ and a $\text{CA}(10; 3, 5, 2)$, where $V = \{0, 1\}$. In $\text{CA}(10; 3, 5, 2)$, if we choose columns 1, 3 and 4, the projection of these columns on row 1 produces $\langle 0, 0, 0 \rangle$. Every element of $\{0, 1\}^3$ can be produced at least once from row 1 to row 10.

⁶Private communication from Marijn Heule.

	1	2	3	4
1	0	0	0	0
2	0	0	1	1
3	0	1	0	1
4	0	1	1	0
5	1	0	0	1
6	1	0	1	0
7	1	1	0	0
8	1	1	1	1

(a)

	1	2	3	4	5
1	0	0	0	0	0
2	0	0	0	1	1
3	0	0	1	0	1
4	0	1	0	0	1
5	0	1	1	1	0
6	1	0	0	0	1
7	1	0	1	1	0
8	1	1	0	1	0
9	1	1	1	0	0
10	1	1	1	1	1

(b)

Figure 21.1. (a) a CA(8; 3, 4, 2) and (b) CA(10; 3, 5, 2)

Covering arrays have been found useful in designing tests for software and hardware testing [CDFP97]. For instance, to test a circuit of k Boolean inputs, if the cost of testing all 2^k possible cases is too high, we may want to test only a small number of cases, but ensure that for any t inputs, $t < k$, all the 2^t cases of these t inputs are tested. A covering array CA($n; t, k, 2$) can serve this purpose by regarding each row of the array as a testing case. To reduce the cost of testing, we wish that n is as small as possible. A covering array is optimal if it has the smallest possible number of rows. This number is the *covering array number*,

$$\text{CAN}(t, k, v) = \min n : \exists \text{CN}(n; t, k, v).$$

A great challenge in design theory is to decide the value of $\text{CAN}(t, k, v)$. Over the last decade, there has been a large body of work done in this field of design theory (see [Har02] for a survey). Other applications related to covering arrays include authentication [WC81], block ciphers [KJS97], data compression [SMM98], intersecting codes [CZ94], etc. Other names used in the literature for covering arrays are *covering suites*, (k, v) -*universal sets*, and t -*surjective arrays*.

The following general results are known about the lower and upper bounds of $\text{CAN}(t, k, v)$.

Theorem 21.2.12. [Ce96, CDFP97]

- (a) $v^t \leq \text{CAN}(t, k, v) \leq v^k$;
- (b) if $j < k$, then $\text{CAN}(t, j, v) < \text{CAN}(t, k, v)$;
- (c) if $u < v$, then $\text{CAN}(t, k, u) < \text{CAN}(t, k, v)$;
- (d) if $s < t$, then $\text{CAN}(s, k, v) < \text{CAN}(t, k, v)$;
- (e) $\text{CAN}(t, k, v) \leq v \text{CAN}(t + 1, k + 1, v)$;
- (f) $\text{CAN}(t, k, vw) \leq \text{CAN}(t, k, v) \cdot \text{CAN}(t, k, w)$.

The above results are hardly useful to deduce the exact values of $\text{CAN}(t, k, v)$. However, *orthogonal arrays*, another subject of design theory, have been found useful to decide some of these numbers.

0000	
0111	0 1 2
0222	2 0 1
1021	1 2 0
1102	
1210	0 1 2
2012	1 2 0
2120	2 0 1
2201	
(a)	(b)

Figure 21.2. (a) an $\text{OA}(3; 4, 2, 1)$; and (b) a pair of $\text{MOLS}(3)$

21.2.5. Orthogonal Arrays

Orthogonal arrays are structures that have been used in the design of experiments for over 50 years. An *orthogonal array*, $\text{OA}(v; k, t, \lambda)$, with degree k (or k constraints), order v (or v levels), strength t , and index λ is an $n \times k$ matrix on v symbols, where $n = \lambda v^t$, with the property that for any t columns, each of v^t vectors of length t on the v symbols appears exactly λ times as the projection of a row on the t columns. It is clear that an orthogonal array of index $\lambda = 1$ is a special case of a covering array since in a covering array, each t -vector is required to appear at least once. In fact, an orthogonal array of index 1 is always a minimal covering array.

In the literature, orthogonal arrays of strength $t = 2$ and index $\lambda = 1$ are extensively studied. In this case, an $\text{OA}(v; k, 2, 1)$ is equivalent to $k - 2$ mutually orthogonal Latin squares (MOLS) of order v . Suppose L_1, L_2, \dots, L_m are a set of MOLS over $V = \{0, 1, \dots, (v - 1)\}$ and we define a $v^2 \times (m + 2)$ table, each row of which is

$$\langle i, j, L_1(i, j), L_2(i, j), \dots, L_m(i, j) \rangle$$

where $0 \leq i, j \leq v - 1$. Then such a table is an $\text{OA}(v; m + 2, 2, 1)$ [Ce96], which is also a minimal $\text{CA}(v^2; 2, m + 2, v)$. Figure 21.2.5 gives an $\text{OA}(3; 4, 2, 1)$ and the corresponding two $\text{MOLS}(3)$.

When looking for the existence of orthogonal arrays, the concept of so-called *incomplete* orthogonal array with index λ have been proved to be useful in the singular product construction of large orthogonal arrays and other constructions of various combinatorial designs. An *incomplete orthogonal array* $\text{OA}(v, u; k, t, \lambda)$ over an alphabet V , $|V| = v$, with a hole $H \subset V$, $|H| = u$, degree k , order v , hole size u , strength t , and index λ is an $n \times k$ matrix on V , where $n = \lambda(v^t - u^t)$, with the property that for any t columns, each vector in $V^t - H^t$ appears exactly λ times as the projection of a row on the t columns.

Theorem 21.2.13. [ACYZ97] *An $\text{OA}(v, u; 5, 2, \lambda)$ exists if and only if $v \geq 4u$, with one exception of $(v, u, \lambda) = (6, 1, 1)$, and three possible exceptions of $(v, u) = (10, 1)$, $(48, 6)$ and $(52, 6)$.*

Note that an incomplete $\text{OA}(v, 1; k, t, \lambda)$ is the same as an ordinary $\text{OA}(v; k, t, \lambda)$ and the existence of $\text{OA}(10, 1; 5, 2, 1)$ implies the existence of three $\text{MOLS}(10)$ (see section 21.2.1.2).

Results obtained by model generators

SATO has been used to establish several cases in Theorem 21.2.13: They are $OA(v, u; 5, 1, 1)$ for $(v, u) \in \{(20, 3), (26, 3), (30, 3), (32, 3), (22, 5), (28, 5), (34, 5)\}$ [ACYZ97].

21.2.5.1. Covering Array Numbers of Small Strength

Charlie Colbourn is maintaining a list of lower bounds for hundreds of covering array numbers of strength from 2 to 6 [Col08]. Some of these results are obtained from orthogonal arrays, some from theoretic work in design theory, some from other search methods, such as greedy search, tabu search, or simulated annealing. Special software has been created for covering arrays of small sizes [YZ08]. At this moment, only one result, i.e., $CAN(3, 7, 3) \leq 40$, is obtained by a SAT-based model generator [HPS04].

Recall that $N(v)$ is the maximum number of Latin squares in a set of MOLS of order v (see section 21.2.1.2). The following result is useful for $CAN(2, k, v)$.

Theorem 21.2.14. *For any $2 \leq k \leq N(v) + 2$, $CAN(2, k, v) = v^2$.*

Proof From $N(v)$ MOLS, we may build an $OA(v^2; N(v) + 2, v, 2, 1)$ which is also $CA(v^2; 2, N(v) + 2, v)$, so $CAN(2, N(v) + 2, v) \leq v^2$. By Theorem 21.2.12 (a), $CAN(2, k, v) \geq v^2$ for any k . So $CAN(2, N(v) + 2, v) = v^2$. By Theorem 21.2.12 (b), the theorem holds.

For unknown $N(v)$ or $k \geq N(v) + 3$, $CAN(2, k, v)$ are less known. We have the following results.

Theorem 21.2.15. (Metsch’s Theorem) *$CAN(2, k, v) > v^2$ for all k satisfying one of the following conditions:*

1. $k > n - 4$ and $n = 14, 21, 22$,
2. $k > n - 5$ and $n = 30, 33, 38, 42, 46, 52, 57, 62, 66, 69, 70, 77, 78$, or
3. $k > n - 6$ and $n = 86, 93, 94$.

However, these lower bounds are of little use in that they do not tell us how fast $CAN(2, k, v)$ grows as a function of k . The only case when $CAN(2, k, v)$ can be defined as a function of k is when $v = 2$.

Theorem 21.2.16. *For all $k > 1$, $CAN(2, k, 2) = n$, where n is the smallest integer satisfying*

$$k \leq \binom{n - 1}{\lceil n/2 \rceil}$$

The above theorem yields the following precise values for $CAN(2, k, 2)$ [Har02].

k	2 – 3	4	5 – 10	11 – 15	16 – 35	36 – 56	57 – 126
n	4	5	6	7	8	9	10

In Tables 21.3 and 21.4, we list the latest known values of $CAN(t, k, v)$ for small k and v and $t = 2, 3$ [CK02, Col08]. If an entry at row k and column

Table 21.3. Known values for $\text{CAN}(2, k, v)$.

$k \setminus v$	2	3	4	5	6	7	8	9
3	4	9	16	25	36	49	64	81
4	5	9	16	25	37	49	64	81
5	6	11	16	25	37, 39	49	64	81
6	6	12	19	25	37, 41	49	64	81
7	6	12	19, 21	29	37, 42	49	64	81
8	6	12, 13	19, 23	29, 33	39, 42	49	64	81
9	6	12, 13	19, 24	29, 35	39, 48	52, 63	64	81
10	6	12, 14	19, 24	29, 37	39, 52	52, 63	67, 80	81
11	7	12, 15	19, 25	29, 38	39, 55	52, 73	67, 80	84, 120
12	7	12, 15	19, 26	29, 40	39, 57	52, 76	67, 99	84, 120
13	7	12, 15	19, 26	29, 41	39, 58	52, 79	67, 102	84, 120
14	7	12, 15	19, 27	29, 42	39, 60	52, 81	67, 104	84, 131

Table 21.4. Known values for $\text{CAN}(3, k, v)$.

$k \setminus v$	2	3	4	5	6	7
4	8	27	64	125	216	343
5	10	28, 33	64	125	222, 260	343
6	12	33	64	125	222, 260	343
7	12	36, 45	76, 88	125, 185	222, 314	343
8	12	36, 45	76, 88	145, 185	222, 341	343
9	12	36, 51	76, 112	145, 185	234, 423	343, 510
10	12	36, 51	76, 112	145, 185	234, 455	364, 510
11	12	36, 57	76, 121	145, 225	234, 455	364, 637
12	14, 15	36, 57	76, 121	145, 225	234, 465	364, 637
13	14, 16	36, 69	76, 124	145, 245	234, 524	364, 637
14	14, 16	36, 69	76, 124	145, 245	234, 524	364, 637

v is a single integer a , it means $\text{CAN}(t, k, v) = a$; if it is a, b , it means $a \leq \text{CAN}(2, k, v) \leq b$.

If $v = p^\alpha$ is a prime power, then $N(v) = v - 1$. By Theorem 21.2.16, $\text{CAN}(2, k, v) = v^2$ for $2 \leq k \leq v + 1$. This result has been generalized to any strength $t < v$.

Theorem 21.2.17. [Har02] *Let $v = p^\alpha$ be a prime power and $v > t$. Then $\text{CAN}(t, k, v) = v^t$ for all $2 \leq k \leq v + 1$. Moreover, if $v \geq 4$ is a power of 2, then $\text{CAN}(3, k, v) = v^3$ for all $2 \leq k \leq v + 2$.*

Results obtained by model generators

In [HPS04], Hnich, Prestwich, Selensky, and Smith studied four different ways of specifying the covering array number problem in propositional logic: the naive matrix approach, the alternative matrix approach, the integrated matrix approach, and the weakened matrix approach. Using incomplete SAT solvers, they reproduced many results in the list maintained by Colbourn. They also found an improved bound for a large problem: $\text{CAN}(3, 7, 3) \leq 40$. This result is still the best known today. Their experimental results showed that incomplete

SAT solvers produce the best results in the weakened matrix approach.

21.2.6. Steiner Systems

Given three integers t, k, n such that $2 \leq t < k < n$, a *Steiner system* $S(t, k, n)$ is a pair (V, \mathcal{B}) , where $|V| = n$ and $\mathcal{B} = \{B \subset V : |B| = k\}$ and B is called a *block*, with the property that every subset of V of size t is contained in exactly one block. Unaware of the earlier work of Thomas Kirkman, in 1853 Steiner first asked about the existence of $S(2, 3, n)$ and went on to ask about $S(t, t + 1, n)$ systems.

Steiner systems have a rich relationship with other structures in design theory. For instance, $S(2, k, n)$ is a *balanced incomplete block design*. Let K_n be the complete undirected graph with n vertices. An $S(4, 5, n)$ is equivalent to a collection \mathcal{B} of edge disjoint pentagons which partition the edges of K_n with the additional property that every pair of vertices are joined by a path of length 2 in exactly one pentagon of \mathcal{B} .

For small $t \geq 4$, only finitely many $S(t, k, n)$ are known and none are known for $t \geq 6$. In the following, we will present some results when $k = 3, 4, 5$.

21.2.6.1. Steiner Triple Systems

An $S(2, 3, n)$ is called *Steiner triple system* of order n , noted as STS(n). It is known that an STS(n) exists if and only if $n \equiv 1$ or $3 \pmod{6}$. Kirkman first proved it in 1847 (six years before Steiner posed the question of its existence!).

If (V, \mathcal{B}) is an $S(t, k, n)$ and if there exists a partition R of the block set \mathcal{B} into *parallel classes*, each of which in turn is a partition of the set V , $S(t, k, n)$ is said to be *resolvable* and R is called a *resolution*. A resolvable STS(n) is called *Kirkman triple system* KTS(n). In 1850, Kirkman posed the following problem.

Kirkman's schoolgirl problem. Fifteen young ladies in a school walk of three abreast for seven days in succession: it is required to arrange them daily, so that no two walk twice abreast.

This problem is equivalent to finding a KTS(15), i.e., a resolvable STS(15). Among the eighty non-isomorphic STS(15)s, exactly four are resolvable. However, there are seven non-isomorphic KTS(15)s [Ce96].

It is known after 1970 that a Kirkman triple system of order n exists if and only if $n \equiv 3 \pmod{6}$ [Ce96].

Results obtained by model generators

While the existence problems for STS(n) and KTS(n) are known, they can serve as interesting SAT benchmarks. Hartley used genetic algorithms on an SAT encoding of STS(n) and found its performance unsatisfactory [Har96].

21.2.6.2. Steiner Quadruple Systems

A Steiner system $S(3, 4, n)$ is called *Steiner quadruple system* of order n , i.e., SQS(n). It is known that an SQS(n) exists if and only if $n \equiv 2$ or $4 \pmod{6}$. For resolvable Steiner quadruple systems, the complete existence result was

established in 2005: A resolvable SQS(n) exists if and only if $n \equiv 4$ or $8 \pmod{12}$ [JZ05].

The most famous problem related to Steiner quadruple systems is the so-called *social golfer problem*:

Social Golfer Problem. A golf club has n members and each member plays golf once a week. Golfers always play in groups of 4 and each golfer will play in the same group with any other golfer exactly once. Can you provide a schedule for this club?

This is apparently a generalization of Kirkman's schoolgirl problem. A solution to the problem is a solution of the resolvable Steiner system $S(2, 4, n)$. For instance, when $n = 16$, and the golfers are named by A to P, a schedule of five weeks is given below:

1	ABCD	EFGH	IJKL	MNOP
2	AEIM	BFJN	CGKO	DHLP
3	AFKP	BELO	CHIN	DGJM
4	AGLN	BHKM	CEJP	DFIO
5	AHJO	BGIP	CFLM	DEKN

The number of weeks can be computed as $(n-1)/3$ as each week a golfer plays with three new friends. Apparently, n must be divisible by 4 so that we can have groups of four. If $n-1$ is not divisible by 3, to allow a solution for such n 's, the problem is relaxed by that "no golfer will play in the same group with any other golfer more than once". In this case, an optimal solution is a schedule of $\lfloor (n-1)/3 \rfloor$ weeks. When $n = 32$, it is problem 10 in CSPLIB (<http://www.csplib.org>) and has drawn a lot of attention. A solution of $n = 32$ is first reported by Hao Shen in 1996, and independently solved by Charles Colbourn in 1999 [Col99]. For $n = 36$, an optimal solution has not been reported. The web site created by Ed Pegg Jr.,

http://www.maa.org/editorial/mathgames/mathgames_08_14_07.html

provides useful information regarding the social golfer problem.

Results obtained by model generators

Like Steiner triple systems, Steiner quadruple systems are interesting benchmark problems for SAT solvers. In particular, Ian Gent and Ines Lynce studied different SAT encoding for the social golfer problem and showed that these encodings may improve or degrade search dramatically depending on the solver [GL05].

21.2.6.3. Steiner Pentagon Systems

When $t = 4$ and $k = 5$, $S(4, 5, n)$ is called *Steiner pentagon system* of order n , denoted by SPS(n). It is known [ABZ99] that a Latin square $(Q, *)$ satisfying the three identities, $x * x = x$, QG11 and QG12 is equivalent to a SPS($|Q|$), where

$$\begin{aligned} \text{QG11: } & (y * x) * x = y \\ \text{QG12: } & x * (y * x) = y * (x * y) \end{aligned}$$

Here a block $(x, y, z, u, v) \in \mathcal{B}$ if and only if $x * y = z, y * x = v$, for $x \neq y$ and $x * x = x$ for all $x \in Q$. It can be deduced that $u = y * z = x * v$. That is, a Latin square $(S, *)$ defines blocks $(x, y, x * y, y * (x * y), y * x)$ for $x, y \in S$. An idempotent Latin square associated with a SPS will be denoted by QG10 (i.e., QG10 is the conjunction of QG11 and QG12).

Theorem 21.2.18. [Ce96, Zha97]

(a) A QG10(n) exists if and only if $n \equiv 1$ or $5 \pmod{10}$, except $n = 15$.
 (b) An FQG10(h^n) exists if and only if $n \geq 5$, $n(n-1)h^2 \equiv 0 \pmod{5}$, and if h is odd, then n is odd, except possibly for the following cases:

1. $h \equiv 10 \pmod{20}$, where $h \neq 30$, and $n = 28$;
2. $h = 30$ and $n \in \{7, 12, 18, 19, 22, 23, 24, 27\}$;
3. $h \equiv 1, 3, 7$, or $9 \pmod{10}$, where $h \neq 3$, and $n = 15$;
4. the pairs $(h, n) \in \{(6, 6), (6, 35), (6, 36), (15, 19), (15, 23), (15, 27)\}$.

Results obtained by model generators

Stickel was the first to study QG10 using a model generator. Using DDPP, he found FQG10(2^5) and FQG10(2^6). Later, the author found a dozen cases of FQG10(h^n) using SATO, including the cases when $(h, n) = (2, 10), (2, 11), (3, 15)$ [ABZ99]. These small order designs were very useful in the recursive constructions of FQG10(h^n) for large h and n . For instance, from FQG10(10^5) we can obtain FQG10(2^{25}) by filling five copies of FQG10(2^5) into the five holes of FQG10(10^5).

21.2.7. Mendelsohn Designs

In Steiner systems, a block is a set of elements. If a block is a list (also called *tuple*) of elements, we have the concept of Mendelsohn designs. A cyclic k -tuple (a_1, a_2, \dots, a_k) is defined to be $\{(a_i, a_{i+1}), (a_k, a_1) \mid 1 \leq i \leq k-1\}$. The elements a_i, a_{i+t} are said to be t -*apart* in a cyclic k -tuple (a_1, a_2, \dots, a_k) , where $i+t$ is taken as $1 + ((i+t-1) \bmod k)$.

Given numbers n, k and λ , a *perfect Mendelsohn design* (briefly (n, k, λ) -PMD) is a pair (X, \mathcal{B}) where

1. X is a set of points,
2. \mathcal{B} is a collection of cyclic k -tuples of X (called blocks),
3. every ordered pair of points appears t -apart in exactly λ blocks for $1 \leq t \leq k-1$.

In this section, we are only interested in (n, k, λ) -PMD where $\lambda = 1$, because they are related to Latin squares. That is, a $(n, k, 1)$ -PMD exists if and only if an idempotent Latin square of order n exists that satisfies the following constraint:

$$x_1 * x_2 = x_3, x_2 * x_3 = x_4, \dots, x_{k-2} * x_{k-1} = x_k \Rightarrow x_{k-1} * x_k = x_1 \quad (21.12)$$

By fixing $k = 3, 4, 5$ in constraint 21.12 and then remove some variables, we obtain the constraints QG13, QG14, and QG5, for $(n, k, 1)$ -PMD, $k = 3, 4, 5$, respectively.

$$\text{QG13: } y * (x * y) = x$$

$$\text{QG14: } (x * y) * (y * (x * y)) = x$$

$$\text{QG15: } (y * (x * y)) * ((x * y) * (y * (x * y))) = x$$

A Latin squares satisfying QG13, QG14, and QG15 correspond to, respectively, perfect Mendelsohn triple, quadruple, pentagon systems.

It is well-known that a QG13(n) exists if and only if $n \equiv 0$ or $1 \pmod{3}$. So people are interested in some combined constraints. For instance, since any conjugate of a QG13(n) also satisfies QG13, if QG13(n) also satisfies QG0, then we have a pair of orthogonal Latin squares, i.e., $(S, *)$ and $(S, *_{213})$, that satisfy QG13. Such a Latin square corresponds to a self-orthogonal perfect Mendelsohn system.

Theorem 21.2.19. [Ce96, Zha97]

(a) *A Latin square of order n satisfying both QG0 and QG13 exists if and only if $n \equiv 0$ or $1 \pmod{3}$, except $n = 3, 6, 9, 10, 12$ and except possibly $n = 18$.*

(b) *A pair of orthogonal QG13(n) exists if and only if $n \equiv 0$ or $1 \pmod{3}$, except $n = 3, 6, 10$ and except possibly $n = 12$.*

QG14 is conjugate-equivalent to QG4, so any existence result regarding QG4 applies to QG14.

The existence results regarding QG15 are mostly known:

Theorem 21.2.20. [Ce96, Zha97] *A QG15(n) exists if and only if $n(n-1) \equiv 0 \pmod{5}$, except $n = 6, 10$ and except possibly $n = 15$ and 20 .*

Results obtained by model generators

For the Latin squares satisfying QG0 and QG13, SATO solved 11 cases in Theorem 21.2.19, 21 being the smallest and 72 being the largest [BZZ96]. McCune is the first to find a pair of orthogonal QG13(9)'s. Later, SATO proved that there exist no pairs of orthogonal QG13(10)'s. SATO also found a pair of orthogonal QG13(18). In the proof of Theorem 21.2.20, SATO found five new cases [BCYZ97, BYZ98].

21.2.8. Magic Squares

A *magic square* of order n , $M = (a_{i,j})$, $0 \leq i, j \leq n-1$, is an $n \times n$ matrix of n^2 distinct nonnegative integers with the property that the sum of the n numbers in every row, in every column and in each diagonal is the same. The sum is called the *magic number*.

The study of magic squares probably dates back to prehistoric times [And60]. A lot of work had been done on the constructions of magic squares, and the interested readers may refer to [Abe94, Cam73, KM06]. Recently, magic squares with some properties are investigated, see [CCZ15, WL12, ZCWZ17].

Given any positive integer n , let $Z_n = \{0, 1, \dots, n-1\}$. A magic square M of order n is called *normal* if the entries of M are the members of Z_{n^2} . In the following, we consider only normal magic squares.

In [KM06], the concept of *symmetric magic square* is introduced. A magic square $M = (m_{i,j})$ of order n is called *symmetric*, SMS(n) for short, if

$$m_{i,j} + m_{n-1-i,n-1-j} = n^2 - 1, \text{ for all } 0 \leq i, j \leq n - 1.$$

In [Gar98], it is shown that if there exists a symmetric magic square of order n , then $n \neq 4k + 2$ for any positive integer k .

For any magic square M of order n over the set Z_{n^2} , we can factor M into (A, B) , where $A = (a_{i,j})$, $B = (b_{i,j})$ are two $n \times n$ matrices over the set Z_n , such that $M = nA + B$, i.e., $m_{i,j} = na_{i,j} + b_{i,j}$, $0 \leq a_{i,j}, b_{i,j} \leq n - 1$, and $M = (m_{i,j}) = nA + B$. It is not difficult to check that A and B are orthogonal if M is a normal magic square [Abe94].

Following the definitions from [Abe94], M is called *elementary* if both A and B are Latin squares; otherwise, M is said to be *nonelementary*. If both A and B possess the property that the sums of the n numbers in every row and every column are the same, then M is called *rational*; otherwise, M is said to be *irrational*. It is clear that an elementary magic square is also rational, an irrational magic square is also nonelementary. The concept of elementary, nonelementary, irrational was introduced in [Abe94] in 1994, however, they did not get enough attention.

In [GS04], streamlining constraints were introduced for constraint reasoning. To see the effectiveness of streamlining constraints, the concept of *diagonally ordered magic square* is introduced. A magic square $M = (m_{i,j})$ of order n is called *diagonally ordered*, DOMS(n) for short, if both the main diagonal and the back diagonal, when traversed from left to right, have strictly increasing values, i.e.,

$$m_{i,i} < m_{i+1,i+1} \quad \text{and} \quad m_{n-1-i,i} < m_{n-2-i,i+1} \quad \text{for all } 0 \leq i \leq n - 2.$$

Here are the existence results regarding normal magic squares with various properties:

Theorem 21.2.21. [ZCWZ17, ZCWZar, YWZ16, YWZar]

(a) *There exists an elementary symmetrical DOMS(n) if and only if $n \equiv 0, 1, 3 \pmod{4}$.*

(b) *There exists an elementary DOMS(n) for or each positive integer n with two exceptions of $n = 2, 6$ and two possible exceptions of $n = 22, 26$.*

(c) *There exists a nonelementary rational symmetric DOMS(n) if and only if $n \equiv 0, 1, 3 \pmod{4}$, and $n \geq 4$.*

(d) *There exists a nonelementary rational DOMS(n) if and only if $n \neq 2, 3$.*

(e) *There exists an irrational symmetric DOMS(n) if and only if $n \equiv 0, 1, 3 \pmod{4}$, and $n \geq 5$.*

(f) *There exists an irrational DOMS(n) if and only if $n \geq 5$.*

Results obtained by model generators

Model generators played an important role in Theorem 21.2.21 by finding designs of small orders for all the six cases. Designs of large orders are then

obtained from recursive construction techniques [ZCWZ17, ZCWZar, YWZ16, YWZar].

Since every magic square of order n can be factored into a pair of orthogonal matrices (A, B) over Z_n , we may use model generators to search for a pair of orthogonal (A, B) over Z_n instead of M directly. The elementary, non-elementary, rational and irrational properties of M are defined from (A, B) . All other properties on M can be defined similarly over (A, B) . To ensure that M is symmetric, we require that both A and B be *strongly symmetric*: An $n \times n$ matrix $A = (a_{i,j})$ of order n is called *strongly symmetric* if $a_{i,j} + a_{n-1-i, n-1-j} = n-1$, $0 \leq i, j \leq n-1$. To ensure that M is a diagonally ordered magic square, let $A = (a_{i,j})$ and $B = (b_{i,j})$ for $0 \leq i, j \leq n-1$, we require that $(a_{i,i}, b_{i,i}) < (a_{i+1, i+1}, b_{i+1, i+1})$, and $(a_{n-i-1, i}, b_{n-i-1, i}) < (a_{n-i-2, i+1}, b_{n-i-2, i+1})$, $0 \leq i < n-1$, by the natural lexicographical ordering.

In search for a pair of orthogonal matrices (A, B) , we may ask that the matrix B is the transpose of A , that is, A is self-orthogonal. For example, for the order $n = 4$, we found the following A by model generators:

$$A = \begin{pmatrix} 0 & 2 & 1 & 3 \\ 3 & 1 & 2 & 0 \\ 3 & 1 & 2 & 0 \\ 0 & 2 & 1 & 3 \end{pmatrix}, \quad B = \begin{pmatrix} 0 & 3 & 3 & 0 \\ 2 & 1 & 1 & 2 \\ 1 & 2 & 2 & 1 \\ 3 & 0 & 0 & 3 \end{pmatrix}, \quad M = \begin{pmatrix} 0 & 11 & 7 & 12 \\ 14 & 5 & 9 & 2 \\ 13 & 6 & 10 & 1 \\ 3 & 8 & 4 & 15 \end{pmatrix}$$

Here the sum of each row, each column and each diagonal of A is the same, i.e., 6. $B = A^T$ and $M = 4A + B$ is a nonelementary rational DOM(4). In this way, we obtained the same type of structures by model generators for order $n = 5, 6, 7, 9, 11, 13, 15, 19$, and 23, thus obtained the corresponding DOM(n).

21.3. Encoding Design Theory Problems

If we regard an off-the-shelf general-purpose SAT solver in a Mace-style model generator as the “hardware”, then the design of the encoder, that is, how to encode these problems for SAT solvers, would be the “software” issues. Indeed, the encoding is one of the most crucial issues to the success of solving these combinatorial problems. In this section, we will discuss some general issues of how to specify combinatorial designs in propositional logic. We examine performance enhancing techniques used in the study, such as adding extra constraints, eliminating isomorphism, etc.

21.3.1. Transforming Specification into Clauses

Most design theory problems that could be attacked by a Mace-style model generator can be specified by a first-order formula over a finite domain. Transforming the first-order formula into propositional clauses is often straightforward but also very important to the success of model generators. First, the transformation must be sound so that the first-order formula is satisfiable if and only if the propositional clauses are satisfiable. Secondly, the clause size should be as small as possible in terms of numbers of literals.

We assume the conventional concepts of a first-order language such as predicates, (non-Boolean) functions (including constants), variables, terms, atoms, clauses, etc. Let u, \dots, z denote free variables, s, t denote terms, and c denote constants. Let S be a set of first-order clauses over a finite domain D , where $D = \{0, 1, \dots, (n-1)\}$. For each predicate p of arity k , we define a propositional variable for every $p(d_1, \dots, d_k)$, where $d_1, \dots, d_k \in D$. For each function of arity k , we define a propositional variable for every equality $f(d_1, \dots, d_k) = d$, where $d_1, \dots, d_k, d \in D$. The transformation of S into a set of propositional clauses can be summarized as the following rules [CS03]:

- **instantiation**

$$\frac{S \cup \{C[x]\}}{S \cup \{C[x/0], C[x/1], \dots, C[x/(n-1)]\}}$$

where $C[x]$ denotes a clause containing the variable x and $C[x/d]$ denote the clause obtained from C by replacing every occurrence of x by d .

- **flattening**

$$\frac{S \cup \{C \vee L[t]\}}{S \cup \{(x \neq t) \vee C \vee L[t/x]\}}$$

where $L[t]$ denotes a literal containing a term t , and t is neither a variable nor an element of D ; L is neither of form $t = s$ nor $t \neq s$, where s is either a variable or an element of D ; x is a fresh variable. $L[t/x]$ represents the literal obtained from L by replacing t by x .

- **term definition**

$$\frac{S \cup \{C \vee L[t]\}}{S \cup \{(c = t), C \vee L[t/c]\}}$$

where t is a non-constant ground term and contains at least one constant other than elements of D ; L is neither of form $t = s$ nor $t \neq s$, where s is either a variable or a constant, or an element of D ; c is a fresh constant. The same c can be reused for all the occurrence of t in S .

- **splitting**

$$\frac{S \cup \{C[\mathbf{x}] \vee D[\mathbf{y}]\}}{S \cup \{C[\mathbf{x}] \vee p(\mathbf{x} \cap \mathbf{y}), D[\mathbf{y}] \vee \neg p(\mathbf{x} \cap \mathbf{y})\}}$$

where \mathbf{x} and \mathbf{y} are the set of variables, occurring in C and D , respectively; neither $\mathbf{x} - \mathbf{y}$ nor $\mathbf{y} - \mathbf{x}$ is empty; p is a fresh predicate.

- **functional definition**

$$\frac{S}{S \cup \{f(d_1, \dots, d_k) \neq d \vee f(d_1, \dots, d_k) \neq d'\}}$$

where f is a function of arity k , and $d_1, \dots, d_k, d, d' \in D$. This rule applies to every function once and for all.

- **totality definition**

$$\frac{S}{S \cup \{f(d_1, \dots, d_k) = 0 \vee \dots \vee f(d_1, \dots, d_k) = (n-1)\}}$$

where f is a function of arity k , and $d_1, \dots, d_k \in D$. This rule applies to every function once and for all.

Let $S \Rightarrow S'$ if $S' \neq S$ and S' is derived from S by one of the above rules and let \Rightarrow^+ be the transitive closure of \Rightarrow . We also write $S \Rightarrow^+ S'$ if $S \Rightarrow^+ S'$ and there exists no $S'' \neq S'$ such that $S' \Rightarrow S''$.

Theorem 21.3.1. [CS03]

(a) \Rightarrow^+ is terminating.

(b) If $S \Rightarrow^+ S'$ then every clause of S' corresponds to a single propositional clause.

(c) If $S \Rightarrow^+ S'$ then S is satisfiable in the domain D if and only if the set of propositional clauses corresponding to S' is satisfiable.

The reader may consult [CS03] for a proof of the above theorem. In the above rules, the **splitting** and **term definition** rules are optional. However, they can help to reduce the number of clauses substantially, as illustrated next by two examples.

If a clause has k variables, the **instantiation** rule will generate n^k ground clauses, where n is the domain size. If each ground clause cannot be represented by a single propositional clause, then the **flattening** rule can be used first which introduces new variables. For instance, for the QG3 constraint in section 21.2.1.5,

$$\text{QG3: } (x * y) * (y * x) = x$$

The **flattening** rule can be used twice to generate

$$(x * y) \neq w \vee (y * x) \neq z \vee (w * z) = x$$

and the **instantiation** rule would generate n^4 ground clauses, each of which corresponds to a propositional clause.

Suppose we are given an identity $(x * y) * (y * a) = x$ where a is a constant. If we follow the same procedure, we will get

$$(u \neq a) \vee (x * y) \neq w \vee (y * u) \neq z \vee (w * z) = x$$

which would generate n^5 propositional clauses. On the other hand, if we apply the **instantiation** rule on y first in $(x * y) * (y * a) = x$, we get n instances of form $(x * d) * (d * a) = x$, where $d \in D$. Applying the **term definition** rule on $(d * a)$, we have two clauses: $c = (d * a)$ and $(x * d) * c = x$. The flattened clause of $c = (d * a)$ is $c \neq z \vee a \neq y \vee z = (d * y)$ which generates n^2 propositional clauses; the flattened clause of $(x * d) * c = x$ is $c \neq z \vee (x * d) \neq y \vee y * z = x$, which generates n^3 propositional clauses. Altogether, $n^4 + n^3$ propositional clauses will be generated. In other words, using the **term definition** rule n times (with n new constants), we reduce the number of propositional clauses from n^5 to $n^4 + n^3$.

In the above example, the **instantiation** rule was used once before the **term definition** rule. In the Paradox system of Claessen and Sörensson [CS03], the **instantiation** rule is the last rule to be used, thus the reduction of clauses illustrated in the above example is not available in Paradox. On the other hand, if we always use the **instantiation** rule first, then it may not be possible to use the **splitting** rule, which is very important as illustrated in the next example. Note also that the **term definition** rule should be used before the **flattening** rule. It

is an interesting research problem to develop useful heuristics to decide the order of using these rules.

For the second example, let us take a look at

$$x * y \neq s \vee z * w \neq s \vee x \star y \neq t \vee z \star w \neq t \vee x = z \quad (21.8)$$

in section 21.2.1.2, which comes from the orthogonality of two matrices. We need to convert this clause into propositional clauses and it is expensive to use only the **instantiation** rule as it contains 6 variables. However, applying the **splitting** rule twice, we obtain three clauses with only four free variables each.

At first, the above clause is split into

$$x * y \neq s \vee x \star y \neq t \vee x = z \vee p(z, s, t), \quad (21.13)$$

$$z * w \neq s \vee z \star w \neq t \vee \neg p(z, s, t) \quad (21.14)$$

where $p(z, s, t)$ is a new predicate. Then clause (21.13) is split into

$$x * y \neq s \vee x \star y \neq t \vee q(x, s, t), \quad (21.15)$$

$$x = z \vee p(z, s, t) \vee \neg q(x, s, t) \quad (21.16)$$

where $q(x, s, t)$ is a new predicate.

In [Zha97], clause (21.8) is manually split into two clauses as shown by the following lemma.

Lemma 21.3.2. ([Zha97]) *For any set C of clauses, $C \cup \{(21.8)\}$ is satisfiable if and only if $C \cup \{(21.17), (21.18)\}$ is satisfiable, where*

$$x * y = s \wedge x \star y = t \Rightarrow \Phi(x, s, t), \quad (21.17)$$

$$\Phi(x, s, t) \wedge \Phi(z, s, t) \Rightarrow x = z, \quad (21.18)$$

and Φ is a new predicate not appearing in C .

As reported in [Zha97], quite a number of quasigroup problems could not be solved without using the new predicate Φ . To attack the open problem of deciding if $N(10) \geq 3$ in section 21.2.1.2, we need to specify three Latin quares of size 10, which requires 3000 propositional variables. To use the above lemma, we need another 3000 propositional variables to specify $\Phi(x, s, t)$. Even though the number of clauses is over 85,000, it is much smaller than 10^6 when without Φ . It is a challenge to develop a method which can automatically generate the predicates like Φ which involves more than one literal in a clause.

21.3.2. Isomorphic Model Elimination

For most design theory problems, including all the problems presented in the previous sections, one solution may generate another by switching names of elements, switching rows or columns in a table, etc. In other words, there exist many isomorphic models satisfying the same constraints. In any model generator based on exhaustive search, it is often too expensive to search all the possible models. Thus isomorphism elimination is crucial to the efficiency of model generators.

Let us first take a look at how numerous Latin squares for some small orders. Two quasigroups $(S, *)$ and (S, \star) are said to be *isomorphic* if there is a bijection $\phi : S \rightarrow S$ such that $\phi(x*y) = \phi(x)\star\phi(y)$. $(S, *)$ and (S, \star) are said to be *isotopic* (or *equivalent*) if there exist three bijections from the rows, columns, and symbols of $(S, *)$ to the rows, columns, and symbols, respectively, of (S, \star) , that map $(S, *)$ to (S, \star) . A Latin square is said to be *reduced* if the first row and the first column contain the elements in natural order. The following table gives the numbers of isotopic, isomorphic, and reduced Latin squares of order less than nine.

$n =$	1	2	3	4	5	6	7	8
isotopy classes	1	1	1	2	2	22	563	1,676,267
isomorphism classes	1	1	5	35	1,411	1,130,531	12,198,455,835	$> 1.52 \times 10^{22}$
reduced squares	1	1	1	4	56	9,408	16,942,080	535,281,401,856

Many people realized the importance of effectively detecting symmetries in a set of propositional clauses [BS94, CGLR96, ARMS03]. Fujita, Slaney and Stickel used a single clause to eliminate some isomorphic Latin squares [SFS95]. To search a Latin square $(S, *)$, where $S = \{0, 1, \dots, (n-1)\}$, the single clause to be added to the input is:

$$y > (x+1) \Rightarrow (x*0 \neq y) \quad (21.19)$$

where $0 \leq x < y < n$ for a Latin square of order n . When converting into SAT clauses, this clause generates only unit clauses. This approach is simple and effective in reducing search time no matter whether the instance is satisfiable or not. In [Zha97], various ways of using similar clauses like the above one are discussed.

Deciding the exact number of non-isomorphic Latin squares satisfying certain constraints is a computationally expensive and interesting problem. For instance, only recently, it is found that there are only six non-isomorphic QG4(9). For quasigroups of orders bigger than 9 and for other types of short identities, we know many of them exist, but we do not know the total number of non-isomorphic ones.

Jian Zhang used the so-called least number heuristic (LNH) [Zha91] in his program FALCON [Zha96] and later in SEM [ZZ95] to avoid exploring isomorphic subspaces. It is based on the observation that, at the early stage of the search, many value names (which have not yet been used in the search) are symmetric. When choosing a value for a variable of the constraints, we need not consider all the values of a given sort. It makes no difference to assign the value e or e' to the variable, if neither of them has been assigned to some other variables. The so-called *least number heuristic* used in FALCON and SEM is that when all the unused values for a variable are symmetric for the constraints, you need only the least one, plus all the used values, for the next variable [Zha96].

The least number heuristic has been shown very effective in reducing isomorphic subspaces in Latin square problems as well as in many other design theory problems. However, this technique in general cannot be fully used without modifying model generation programs. For SAT-based (i.e., Mace-style) model generators, an off-shelf SAT solver is often used, and in such case, it is very difficult

to use techniques like the least number heuristic because it involves the internal working of SAT solvers. Claessen and Sörensson [CS03] investigated the way of using the idea of the least number heuristic in a Mace-style model generator. For constant symbols (other than the ones introduced by **term definition**, they can be assigned the least numbers in the domain. Let $k > 0$ be the number of constants and f be a function of arity 1. They suggest the following clauses which subsume the *totality definitions* of f :

$$\begin{aligned} f(0) = 0 \vee f(0) = 1 \vee \dots \vee f(0) = k \\ f(1) = 0 \vee f(1) = 1 \vee \dots \vee f(1) = k \vee f(1) = k + 1 \\ f(2) = 0 \vee f(2) = 1 \vee \dots \vee f(2) = k \vee f(2) = k + 1 \vee f(2) = k + 2 \\ \dots \end{aligned}$$

The rationale here is that, in order to pick a value for a particular $f(d)$, we simply use an element that we have already used, or the least element that has not been used yet. A more efficient way would be to generate only unit clauses. That is, suppose the number of elements is n , we generate the following unit clauses:

$$\begin{aligned} f(0) \neq k + 1, f(0) \neq k + 2, \dots, f(0) \neq n - 1 \\ f(1) \neq k + 2, f(1) \neq k + 3, \dots, f(1) \neq n - 1 \\ f(2) \neq k + 3, f(2) \neq k + 4, \dots, f(2) \neq n - 1 \\ \dots \end{aligned}$$

These unit clauses can not only generate the previous set of clauses with the totality definitions, but also subsume many other input clauses.

For function g of arity greater than 1, Claessen and Sörensson suggested to define a fresh function f in terms of g by e.g. $f(x) = g(x, 0)$. In this case, the effect is the same as the isomorphic cutting clause used by Fujita, Slaney and Stickel.

Of course, it is possible to encode fully the idea of the least number heuristic for any function as a set of propositional clauses by introducing the canonicity requirement [CS03]. However, adding a large set of clauses to the original set often leads to a longer search time. The same experience has been shared by other researchers. For instance, to search for lower bounds of Van der Waerden numbers, Dransfield, Marek and Truszczynski [DMT04] have tried to add extra clauses to eliminate some isomorphic models. However, adding such clauses hurt the performance of their incomplete SAT solver. Gent and Lynce [GL05] and Azevedo [Aze06] also have tried different ways of removing isomorphic models in the study of the social golfer problem. Depending on the types of SAT solvers, the performance varies.

The symmetries of these design theory problems are easy to detect in their specification. As the clauses converted from the specification is often numerous, it is much harder to detect symmetries in the propositional clauses, because checking the symmetry of a set of propositional clauses itself is also NP-complete [Cra92]. It remains a challenge to develop a generic tool which takes the specification of a design theory problem and produces powerful isomorphic-cutting clauses. Huang et al. [HMG⁺18] used the least number heuristic implemented in SEM to generate many SAT instances which are subproblems of the large set problems in Theorem 21.2.10 and solved several previously open cases. It appears that this

approach of combining model generators like SEM with SAT solvers is effective and easy to use.

21.3.3. Satisfiability modulo SAT

In section 21.2.8, we introduced the concept of magic squares with various constraints. To avoid large numbers appearing in the encoding of magic squares for propositional logic, it recommends to represent a magic square by a pair of orthogonal matrices. The existence of a magic square M of order n becomes the existence of (A, B) , where $M = nA + B$, which can be easily specified as an instance of SAT. To search for a rational magic square, we require that the sum of every row, column and diagonal of both A and B is equal to the same number, i.e., $(n - 1)n/2$. This constraint is troublesome for SAT solvers.

Let $p_{x,y,z}$ be the Boolean variable with the meaning that the entry at row x and column y in the matrix A is z , where $0 \leq x, y, z < n$. To specify that the sum of the entries in row x is $(n - 1)n/2$, we may use the following clause:

$$\neg p_{x,0,z_0} \vee \neg p_{x,1,z_1} \vee \cdots \vee \neg p_{x,n-1,z_{n-1}}$$

where $0 \leq x, z_i < n$, and $z_0 + z_1 + \cdots + z_{n-1} \neq (n - 1)n/2$. Similarly, we may specify that the sum of the entries in each column (and the two diagonals) is the same. The number of propositional clauses represented by the above clause is $O(n^{n+1})$. For $n = 10$, it will be around 10^{11} , far exceeding the limit of today's SAT solvers.

Using the linear pseudo-boolean logic, we may specify that the sum of the entries in row x is $(n - 1)n/2$ as follows:

$$\sum_{y=0}^{n-1} \sum_{z=0}^{n-1} (z + 1)p_{x,y,z} = (n + 1)n/2.$$

For $n = 10$, there are 100 variables in the above formula and this large number of variables is a source of inefficiency for pseudo-boolean solvers.

Using the so-called satisfiability modulo SAT approach proposed in [Zha16], we may create a simple procedure for checking that in A and B , the sum of the entries in each row, column and the two diagonals is the same. Using this decision procedure in a SMT solver, we can apply the advanced techniques of modern SAT solvers to the existence problem of magic squares presented in section 21.2.8. In fact, all the solved cases in section 21.2.8 by a model generator are done by satisfiability modulo SAT.

The idea of satisfiability modulo SAT [Zha16] can be described in the context of the SMT (*satisfiability modulo theories*) approach [NOT06, BSST09], where the interpretation of some propositional variables are constrained by a background theory. For our case, this background theory can be either linear pseudo-boolean logic or simply propositional logic; notably, this background theory can be expressed in propositional logic and easily implemented by a procedure in the framework of the Davis-Putnam-Logemann-Loveland (DPLL) procedure. While this idea does not provide any contribution to the theory of SMT, it is a fresh way

of using the SMT approach and the experiment is a successful practice of a SAT solver based on SMT modulo a procedural theory.

Suppose the SAT instance is described by a set S of propositional clauses and we search for a model of S . In the satisfiability modulo SAT approach, we partition S into Φ and Ψ , and treat Φ as the base constraints and Ψ as a theory. One reason for not treating Ψ as normal clauses is the huge size of Ψ ; it will be much more effective if we use a piece of software code to simulate the effect of Ψ . For instance, the above example of specifying the sum of each row (column, diagonal) of a matrix is equal to the same number, is a good candidate to go to in Ψ . Because propositional logic is well-studied, the completeness and soundness of a decision procedure for Ψ can be easily established. Because of similarity between Φ and Ψ , we may implement an effective SMT procedure which is different from the standard SMT procedure. The details of this approach can be found in [Zha16].

21.4. Conclusions and Open Problems

In this chapter, we have presented various design theory problems where SAT-based model generators are used and some previously-open problems were answered by these model generators. There are still many open problems. Below is a list of open design theory problems presented in the chapter.

1. What is the minimum number of entries for a 9×9 Sudoku puzzle to have a unique solution (section 21.2.1.1)?
2. Are there 3 mutually orthogonal Latin square of size 10 (section 21.2.1.2)?
3. Does SOLSSOM(n) exist for $n = 10$ and 14 (section 21.2.1.2)?
4. Does IQG0(20, 6) exist (section 21.2.1.4)?
5. Does IQG1(11, 3) exist (section 21.2.1.4)?
6. Does IQG2(n, k) exist for $(n, k) = (35, 4)$ and $(38, 4)$ (section 21.2.1.4)?
7. Does FQG1(13^6) exist (section 21.2.1.4)?
8. Does FQG2(h^4) exist for $h = 3, 5$ and 7 (section 21.2.1.4)?
9. Does FQG2(h^4) exist for $h = 3, 5$ and 7 (section 21.2.1.4)?
10. Does QG5(v) exists for $v = 18, 20$ or 22 (section 21.2.1.5)?
11. Does QG6(v) exist for $v = 18, 19, 20$, or 21 (section 21.2.1.5)?
12. Does QG7(v) exist for $v = 18, 19, 20$, or 33 (section 21.2.1.5)?
13. Does QG8(v) exist for $v = 22, 23$, or 26 (section 21.2.1.5)?
14. Does LSQG3(v) exist for $v \in \{12, 13\}$ (section 21.2.1.5)?
15. Does LSQG4(v) exist for $v \in \{9, 12\}$ (section 21.2.1.5)?
16. Does LSQG5(11) exist (section 21.2.1.5)?
17. Does LSQG6(v) exist for $v \in \{9, 13\}$ (section 21.2.1.5)?
18. Does LSQG7(v) exist for $v \in \{9, 13\}$ (section 21.2.1.5)?
19. Does LSQG8(v) exist for $v \in \{9, 11\}$ (section 21.2.1.5)?
20. Does LSQG9(v) exist for $v \in \{10, 13\}$ (section 21.2.1.5)?
21. Is the Ramsey number $R(5, 5)$ greater than 43? Is $R(6, 4)$ greater than 35 (section 21.2.2)?
22. Is the Van der Waerden number $W(2, 7)$ greater than 3704? Is $W(3, 4)$ greater than 293? Is $W(5, 3)$ greater than 171 (section 21.2.3)?

23. Is $\text{CAN}(2, 8, 3) = 12$, $\text{CAN}(2, 7, 4) \leq 20$, or $\text{CAN}(2, 5, 6) \leq 38$ (section 21.2.5.1)?
24. Is $\text{CAN}(3, 12, 2) = 14$, $\text{CAN}(3, 5, 3) \leq 32$, or $\text{CAN}(3, 7, 4) \leq 87$ (section 21.2.5.1)?
25. Can we find a schedule of 11 weeks for 36 golfers in the social golfer problem (section 21.2.6.2)?
26. Does $\text{FQG}10(h^n)$ exist for $(h, n) = (6, 6)$ and $(7, 15)$ (section 21.2.6.3)?
27. Does there exist a self-orthogonal perfect Mendelsohn triple system of order 18 (section 21.2.7)?
28. Do there exist a pair of orthogonal perfect Mendelsohn triple system of order 12 (section 21.2.7)?
29. Does there exist a perfect Mendelsohn pentagon system of order 15 (section 21.2.7)?
30. Does there exist an elementary diagonally ordered magic square of order 22 (section 21.2.8)?
31. What are the exact numbers of non-isomorphic quasigroups of $\text{QG}3(12)$, $\text{QG}4(12)$, and $\text{QG}5(11)$, respectively (section 21.3.2)?

We hope that the developers of model generators will find these design theory problems useful as these problems provide a strong motivation and great challenge for them to develop new techniques to attack them. You will find it how satisfactory and exciting as a researcher if you could solve a problem no one else in the world could.

References

- [Abe94] G. Abe. Unsolved problems on magic squares. *Discrete Math.*, 127:3–13, 1994.
- [ABZ99] R. J. R. Abel, F. Bennett, and H. Zhang. Holey steiner pentagon systems. *J. Combin. Designs*, 7:41–56, 1999.
- [ABZZ00] R. J. R. Abel, F. Bennett, H. Zhang, and L. Zhu. A few more self-orthogonal latin square and related designs. *Austral J. Combin.*, 21:85–94, 2000.
- [ACYZ97] R. J. R. Abel, C. J. Colbourn, J. Yin, and H. Zhang. Existence of incomplete transversal designs with block size five and any index. *Designs, Codes and Cryptography*, 10:275–307, 1997.
- [And60] W. S. Andrews. *Magic Squares and Cubes*. Dover, New York, 1960.
- [ARMS03] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. Solving difficult instances of boolean satisfiability in the presence of symmetry. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(9):1117–1137, 2003.
- [Aze06] F. Azevedo. An attempt to dynamically break symmetries in the social golfers problem. In *Proc. of the 11th Annual ERCIM Workshop on Constraint Solving and Constraint Programming (CSCLP)*, 2006.
- [BCYZ97] F. Bennett, Y. Chang, J. Yin, and H. Zhang. Existence of hpmDs with block size five. *J. Combin. Designs*, 5:257–273, 1997.

- [Ben87] F. Bennett. The spectra of a variety of latin squares and related combinatorial designs. *Discrete Math.*, 34:43–64, 1987.
- [Ber68] E. Berlekamp. A construction for partitions which avoid long arithmetic progressions. *Canadian Math. Bulletin*, 11:409–414, 1968.
- [BS60] R. C. Bose. and S. S. Shrikhande. On the construction of sets of mutually orthogonal latin squares and falsity of a conjecture of euler. *Transactions of the American Mathematical Society*, 95:191–209, 1960.
- [BS94] B. Benhamou and L. Sais. Tractability through symmetries in propositional calculus. *J. Autom. Reasoning*, 12(1):89–102, 1994.
- [BSST09] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. *Satisfiability Modulo Theories*. IOS Press, 2009.
- [BYZ98] F. Bennett, J. Yin, and H. Zhang. Perfect mendelsohn packing designs with block size five. *J. of Designs, Codes and Cryptography*, 14:5–22, 1998.
- [BZ92] F. Bennett and L. Zhu. *Conjugate-orthogonal Latin squares and related structures*. John Wiley & Sons, 1992.
- [BZ98] F. Bennett and H. Zhang. Existence of $(3, 1, 2)$ -hcols and $(3, 1, 2)$ -icoils. *J. of Combinatoric Mathematics and Combinatoric Computing*, (27):53–64, 1998.
- [BZZ96] F. Bennett, H. Zhang, and L. Zhu. Self-orthogonal mendelsohn triple systems. *Journal of Combinatorial Theory*, (A 73):207–218, 1996.
- [Cam73] S. V. R. Cammann. *Magic Squares*. Chicago, 1973.
- [CCZ15] N. Cao, K. Chen, and Y. Zhang. Existence of yang hui type magic squares. *J. Graphs Combin.*, 31:1289–1310, 2015.
- [CDFP97] D. M. Cohen, S. R. Dala, M. L. Fredman, and G. C. Patton. The aetg system: an approach to testing software based on combinatorial design. *IEEE Trans Software Engin.*, 23:437–444, 1997.
- [Ce96] C. J. Colbourn and J. H. D. (eds). *The CRC Handbook of Combinatorial Designs*. CRC Press, 1996.
- [CGLR96] J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Principles of Knowledge Representation and Reasoning (KR'96)*, pages 148–159, 1996.
- [CK02] M. Chateauneuf and D. L. Kreher. On the state of strength-three covering arrays. *J. Combin. Designs*, 10:217–238, 2002.
- [Col99] C. J. Colbourn. A steiner 2-design with an automorphism fixing exactly $r + 2$ points. *Journal of combinatorial Designs*, 7:375–380, 1999.
- [Col08] C. J. Colbourn. Ca tables for $t=2,3,4,5,6$, 2008.
- [Cra92] J. Crawford. A theoretical analysis of reasoning by symmetry in first-order logic (extended abstract). In *AAAI-92 Workshop on Tractable Reasoning*, pages 17–22, San Jose, CA, 1992.
- [CS03] K. Claessen and N. Sörensson. New techniques that improve MACE-style finite model finding. In *CADE-19, Workshop W4. Model Computation — Principles, Algorithms, Applications*, 2003.
- [CZ94] G. Cohen and G. Zemor. Intersecting codes and independent fam-

- ilies. *IEEE Trans Information Theory*, 40:1872–1881, 1994.
- [DD01] O. Dubois and G. Dequen. The non-existence of $(3,1,2)$ -conjugate orthogonal idempotent latin square of order 10. In *Proc. of 7th International Conference on Principles and Practice of Constraint Programming (CP2001)*, 2001.
- [DMT04] M. R. Dransfield, V. W. Marek, and M. Truszczynski. Satisfiability and computing van der waerden numbers. In *Proc. of 6th Intl. Conference on Theory and Applications of Satisfiability Testing*, 2004.
- [ER52] P. Erdős and R. Rado. Combinatorial theorems on classifications of subsets of a given set. *Prod. of London Math. Society*, (2):417–439, 1952.
- [Eva75] T. Evans. Algebraic structures associated with latin squares and orthogonal arrays. *Congr. Numer*, 13:31–52, 1975.
- [FSB93] M. Fujita, J. Slaney, and F. Bennett. Automatic generation of some results in finite algebra. In *Proc. Int. Joint Conf. on Art. Intelligence*, 1993.
- [Gar98] M. Gardner. *Magic squares and cubes, Ch.17 in Time Travel and Other Mathematical Bewilderments*. New York: W. H. Freeman, 1998.
- [GL05] I. P. Gent and I. Lynce. A sat encoding for the social golfer problem. In *IJCAI'05 workshop on Modeling and Solving Problems with Constraints*, 2005.
- [GRS90] R. Graham, B. Rothschild, and J. H. Spencer. *Ramsey Theory*. John Wiley and Sons, NY, 1990.
- [GS02] C. P. Gomes and D. Shmoys. Completing quasigroups or latin squares: A structured graph coloring problem. In *Proceedings of the Computational Symposium on Graph Coloring and Generalizations*, 2002.
- [GS04] C. P. Gomes and M. Sellmann. Streamlined constraint reasoning. In *Lecture Notes in Computer Science*, volume 3258, pages 274–289. Springer-Verlag, 2004.
- [GW99] I. P. Gent and T. Walsh. Csplib: A benchmark library for constraints. Technical Report Technical report APES-09-1999, <http://csplib.org>, 1999.
- [Hal07] G. Halkes. Suvisa, an open source software project. In <http://os.ghalkes.nl/SuViSa.html>, 2007.
- [Har96] S. J. Hartley. Steiner systems and the boolean satisfiability problem. In *Proceedings of the ACM symposium on Applied Computing*, pages 277–281, 1996.
- [Har02] A. Hartman. Software and hardware testing using combinatorial covering suites. *Interdisciplinary Applications of Graph Theory, Combinatorics and Algorithms*, 2002.
- [HFKS02] R. Hasegawa, H. Fujita, M. Koshimura, and Y. Shirai. A model generation based theorem prover mgtp for first-order logic. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, pages 178–213, London, UK, 2002. Springer-Verlag.

- [HHvLvM07] P. Herwig, M. J. H. Heule, M. van Lambalgen, and H. van Maaren. A new method to construct lower bounds for van der waerden numbers. *The Electronic Journal of Combinatorics*, (14), 2007.
- [HMG⁺18] P. Huang, F. Ma, C. Ge, J. Zhang, and H. Zhang. Investigating the existence of large sets of idempotent quasigroups via satisfiability testing. In *Proc. of Intl. Joint Conference on Automated Reasoning (IJCAR-18)*, pages 354–369. Springer Cham, 2018.
- [HPS04] B. Hnich, S. D. Prestwich, and E. Selensky. Constraint-based approaches to the covering test problem. In *Proceedings of the Workshop on Constraint Solving and Constraint Logic Programming, CSCP'04*, volume 3419, pages 172–186. Springer LNAI, 2004.
- [JZ05] L. Ji and L. Zhu. Resolvable steiner quadruple systems for the last 23 orders. *SIAM Journal on Discrete Mathematics*, 19(2):420–430, 2005.
- [KF05] M. Kouril and J. Franco. Resolution tunnels for improved sat solver performance. In *Proc. of Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'05)*, volume 3569. LNCS, 2005.
- [KJS97] K. Kurosawa, T. Johansson, and D. Stinson. Almost k -wise independent sample spaces and their cryptologic applications. In *Advances in Cryptology, Eurocrypt 97, LNCS*, volume 1233, pages 409–421, 1997.
- [KM06] J. M. Kudrle and S. B. Menard. *Magic Squares*. Boca Raton, FL: Chapman and Hall/CRC, 2006.
- [KRS99] S. R. Kumar, A. Russell, and R. Sundaram. Approximating latin square extensions. *Algorithmica*, (24):128–138, 1999.
- [KZ94] S. Kim and H. Zhang. Modgen: Theorem proving by model generation. In *Proc. of National Conference of American Association on Artificial Intelligence (AAAI-94)*, Seattle, WA, pages 162–167. MIT Press, 1994.
- [Lam91] C. W. H. Lam. The search for a finite projective plane of order 10. *American Mathematical Monthly*, 98(4):305–318, 1991.
- [LO06] I. Lynce and J. Ouaknine. Sudoku as a sat problem. In *9th International Symposium on Artificial Intelligence and Mathematics*, 2006.
- [McC94] W. McCune. A davis-putnam program and its application to finite first-order model search: Quasigroup existence problems. Technical Report Technical Report ANL/MCS-TM-194, Argonne National Laboratory, 1994.
- [MZ07] F. Ma and J. Zhang. Computer search for large sets of idempotent quasigroups. In *4th Symposium on Computer and Mathematics*, 2007.
- [NOT06] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving sat and sat modulo theories: From an abstract davis-putnam-logemann-loveland procedure to dpll(t). *Journal of the ACM*, 53:937–977, 2006.
- [Rab79] J. R. Rabung. Some progression-free partitions constructed using folkman's method. *Canadian Mathematical Bulletin*, 22(1):87–91,

- 1979.
- [Rad05] S. Radziszowski. Small ramsey numbers. *The Electronic Journal of Combinatorics*, (1), 2005.
- [Ram30] F. P. Ramsey. On a problem of formal logic. *London Math. Soc.*, 30(series 2):264–286, 1930.
- [Roy07] G. Royle. Minimum sudoku. Technical report, <http://people.csse.uwa.edu.au/gordon/sudokumin.php>, 2007.
- [SFS95] J. Slaney, M. Fujita, and M. Stickel. Automated reasoning and exhaustive search: Quasigroup existence problems. *Computers and Mathematics with Applications*, 29:115–132, 1995.
- [Sla94] J. Slaney. Finder: Finite domain enumerator – system description. In *Proc. CADE-12*, pages 798–801, 1994.
- [SMM98] B. Stevens, L. Moura, and E. Mendelsohn. Lower bounds for transversal covers. *Designs, Codes and Cryptography*, 15(3):279–299, 1998.
- [SZ94] M. Stickel and H. Zhang. First results of studying quasigroup identities by rewriting techniques. In *Proc. of Workshop on Automated Theorem Proving (FGCS'94)*, 1994.
- [WC81] M. Wegman and J. Carter. New hash functions and their use in authentication and set quality. *J. Computing System Science*, (22 (3)):265–279, 1981.
- [Web05] T. Weber. A sat-based sudoku solver. In *Geoff Sutcliffe and Andrei Voronkov, editors, LPAR-12, The 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 11–15, 2005.
- [WL12] F. P. W. Li, D. Wu. A construction for doubly pandiagonal magic squares. *Discrete Math.*, 312:479–485, 2012.
- [YWZ16] H. Yu, D. Wu, and H. Zhang. The existence of irrational diagonally ordered magic squares. *Graphs Combin.*, (32):2159–2170, 2016.
- [YWZar] H. Yu, D. Wu, and H. Zhang. The existence of nonelementary rational symmetric diagonally ordered magic square. *Ars Combin.*, to appear.
- [YZ08] J. Yan and J. Zhang. A backtracking search tool for constructing combinatorial test suites. *J. of Systems and Software*, (2 (34)), 2008.
- [ZB95] H. Zhang and F. Bennett. Existence of some $(3, 2, 1)$ -hcols and $(3, 2, 1)$ -hcpls. *J. of Combin. Math. and Combin. Computing*, 1995.
- [ZBH96] H. Zhang, M. P. Bonacina, and H. Hsiang. Psato: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21:543–560, 1996.
- [ZCWZ17] Y. Zhang, J. Chen, D. Wu, and H. Zhang. Diagonally ordered orthogonal weakly diagonal latin squares and their related elementary diagonally ordered magic squares. *Utilitas Mathematica*, 104:83–102, 2017.
- [ZCWZar] Y. Zhang, J. Chen, D. Wu, and H. Zhang. The existence and application of strongly idempotent self-orthogonal row latin magic arrays. *Acta Mathematica Applicata Sinica, English Series*, to ap-

- pear.
- [Zha91] J. Zhang. Search for idempotent models of quasigroup identities. Technical report, Institute of Software, Academia Sinica, 1991.
 - [Zha96] J. Zhang. Constructing finite algebras with falcon. *J. of Automated Reasoning*, 17(1):1–22, 1996.
 - [Zha97] H. Zhang. *Specifying Latin squares in propositional logic*. MIT Press, 1997.
 - [Zha02] H. Zhang. A random jump strategy for combinatorial search. In *Proc. of Sixth International Symposium on Artificial Intelligence and Mathematics*, 2002.
 - [Zha16] H. Zhang. An experiment with satisfiability modulo sat. *Journal of Automated Reasoning*, 56(2):143–154, 2016.
 - [ZW91] J. Zhang and L. Wos. Automated reasoning and enumerative search, with applications to mathematics. In *Proc. of International Conference for Young Computer Scientists, Beijing*, pages 543–545, 1991.
 - [ZZ95] J. Zhang and H. Zhang. Sem: a system for enumerating models. In *Proc. of International Joint Conference on Artificial Intelligence (IJCAI95)*, pages 11–18, 1995.

Chapter 22

Connections to Statistical Physics

Fabrizio Altarelli, Rémi Monasson, Guilhem Semerjian and Francesco Zamponi

22.1. Introduction

The connection between statistical physics of disordered systems and optimization problems in computer science dates back from twenty years at least [MPV87]. In combinatorial optimization one is given a cost function (the length of a tour in the traveling salesman problem (TSP), the number of violated constraints in constraint satisfaction problems, ...) of (Boolean) variables and attempts to minimize costs. Finding the true minimum may be complicated, and requires more and more computational effort as the number of variables increases [PS98]. Statistical physics is at first sight very different. The scope is to deduce the macroscopic, that is, global properties of a physical system, for instance, whether gas, a liquid or a solid, from the knowledge of the energetic interactions of its elementary components (molecules, atoms or ions). However, at very low temperature, these elementary components are essentially forced to occupy the spatial conformation minimizing the global energy of the system. Hence low-temperature-statistical-physics can be seen as the search for minimizing a cost function whose expression reflects the laws of Nature or, more specifically, the degree of accuracy retained in its description. This problem is generally not difficult to solve for non-disordered systems where the lowest energy conformation are crystals in which components are regularly spaced from each other. Yet the presence of disorder, e.g. impurities, makes the problem very difficult and finding the conformation with minimal energy is a true optimization problem.

At the beginning of the eighties, following the work of G. Parisi and others on systems called spin glasses [MPV87], important progress was made in the statistical physics of disordered systems. The properties of systems given some distribution of the disorder (for instance the location of impurities) such as the average minimal energy and its fluctuations became amenable to quantitative studies. The application to optimization problems was natural and led to beautiful studies on (among others) the average properties of the minimal tour length in the TSP and the minimal cost in Bipartite Matching, for some specific instance distributions [MPV87]. Unfortunately statistical physicists and computer scientists did not establish close ties on a large scale at that time, probably due to

differences of methodological nature [FA86]. While physicists were making statistical statements, true for a given distribution of inputs, computer scientists were rather interested in solving one (or several) arbitrary instances of a problem. The focus was thus on efficient ways to do so, that is, requiring a computational effort growing not too quickly with the number of data defining the instance. Knowing precisely the typical properties for a given, academic distribution of instances did not help much to solve practical cases.

At the beginning of the nineties practitioners in artificial intelligence realized that classes of random constraint satisfaction problems used as artificial benchmarks for search algorithms exhibited abrupt changes of behaviour when some control parameters were finely tuned [MSL92]. The most celebrated example was random k -Satisfiability, where one looks for a solution to a set of random logical constraints over a set of Boolean variables. It appeared that, for a large number of variables, there was a critical value of the number of constraints per variable below which there almost surely existed solutions, and above which solutions were absent. An important feature was that the efficiency of known search algorithms drastically decreased in the vicinity of this critical ratio. In addition to its intrinsic mathematical interest the random k -SAT problem was therefore worth to be studied for ‘practical’ reasons.

This critical phenomenon, strongly reminiscent of phase transitions in the physics of condensed matter, led to a revival of research at the interface between statistical physics and computer science. The purpose of the present review is to introduce the non physicist reader to the concepts required to understand the literature in that field and to present some major results. We shall discuss the refined picture of the satisfiable phase in particular as viewed in statistical mechanics and the impact on the algorithmic approach (Survey Propagation, an extension of Belief Propagation used in communication theory and statistical inference) as inspired by this view.

While the presentation will mostly focus on the k -Satisfiability problem (with random constraints) we will occasionally also discuss the computational problem of solving linear systems of Boolean equations, called k -XORSAT. A good reason to do so is that this problem exhibits tremendous ‘syntactic’ similarities with random k -Satisfiability, while being technically simpler to study. In addition k -Satisfiability and k -XORSAT have very different and interesting computational properties as will be discussed in this chapter. Last but not least k -XORSAT is closely related to error-correcting codes in communication theory.

The chapter is divided into four main parts. In Section 22.2 we present the basic statistical physics concepts related to phase transitions, and their nature. Those are illustrated on a simple example of a decision problem, the so-called perceptron problem. In Section 22.3 we review the scenario of the various phase transitions observable in random k -SAT. Section 22.4 and 22.5 present the techniques used to study various type of algorithms in optimization: Local search, backtracking procedures, and message passing algorithms. We close with some conclusive remarks in Sec. 22.6.

22.2. Phase Transitions: Basic Concepts and Illustration

22.2.1. A simple decision problem with a phase transition: the continuous perceptron

For pedagogical reasons we first discuss a simple example exhibiting several important features we shall define more formally in the next subsection. Consider M points $\underline{T}^1, \dots, \underline{T}^M$ of the N -dimensional space \mathbb{R}^N , their coordinates being denoted $\underline{T}^a = (T_1^a, \dots, T_N^a)$. The continuous perceptron problem consists in deciding the existence of a vector $\underline{\sigma} \in \mathbb{R}^N$ which has a positive scalar product with all vectors linking the origin of \mathbb{R}^N to the \underline{T} 's,

$$\underline{\sigma} \cdot \underline{T}^a \equiv \sum_{i=1}^N \sigma_i T_i^a > 0, \quad \forall a = 1, \dots, M, \quad (22.1)$$

or in other words determining whether the M points belong to the same half-space. The term continuous in the name of the problem emphasizes the domain \mathbb{R}^N of the variable $\underline{\sigma}$. This makes the problem polynomial from worst-case complexity point of view [HKP91].

Suppose now that the points are chosen independently, identically, uniformly on the unit hypersphere, and call

$P(N, M)$ = Probability that a set of M randomly chosen points
belong to the same half-space.

This quantity can be computed exactly [Cov65] (see also Chapter 5.7 of [HKP91]) and is plotted in Fig. 22.1 as a function of the ratio $\alpha = M/N$ for increasing sizes $N = 5, 20, 100$. Obviously P is a decreasing function of the number M of points for a given size N : increasing the number of constraints can only make more difficult the simultaneous satisfaction of all of them. More surprisingly, the figure suggests that, in the large size limit $N \rightarrow \infty$, the probability P reaches a limiting value 0 or 1 depending on whether the ratio α lies, respectively, above or below some 'critical' value $\alpha_s = 2$. This is confirmed by the analytical expression of P obtained in [Cov65],

$$P(N, M) = \frac{1}{2^{M-1}} \sum_{i=0}^{\min(N-1, M-1)} \binom{M-1}{i}, \quad (22.2)$$

from which one can easily show that, indeed,

$$\lim_{N \rightarrow \infty} P(N, M = N\alpha) = \begin{cases} 1 & \text{if } \alpha < \alpha_s \\ 0 & \text{if } \alpha > \alpha_s \end{cases}, \quad \text{with } \alpha_s = 2. \quad (22.3)$$

Actually the analytical expression of P allows to describe more accurately the drop in the probability as α increases. To this aim we make a zoom on the transition region $M \approx N\alpha_s$ and find from (22.2) that

$$\lim_{N \rightarrow \infty} P(N, M = N\alpha_s(1 + \lambda N^{-1/2})) = \int_{\lambda\sqrt{2}}^{\infty} \frac{dx}{\sqrt{2\pi}} e^{-x^2/2}. \quad (22.4)$$

As it should the limits $\lambda \rightarrow \pm\infty$ give back the coarse description of Eq. (22.3)

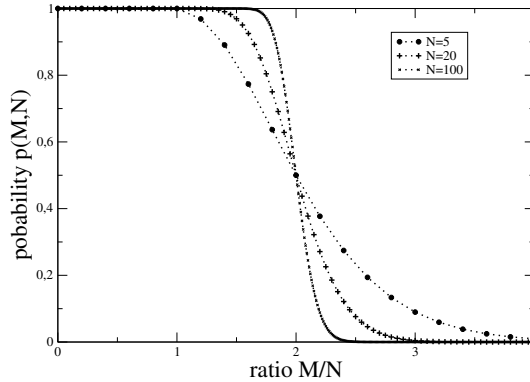


Figure 22.1. Probability $P(N, M)$ that M random points on the N -dimensional unit hypersphere are located in the same half-space. Symbols correspond to Cover’s exact result [Cov65], see Eq. (22.2), lines serve as guides to the eye.

22.2.2. Generic definitions

We now put this simple example in a broader perspective and introduce some generic concepts that it illustrates, along with the definitions of the problems studied in the following.

- Constraint Satisfaction Problem (CSP)

A CSP is a decision problem where an assignment (or configuration) of N variables $\underline{\sigma} = (\sigma_1, \dots, \sigma_N) \in \mathcal{X}^N$ is required to simultaneously satisfy M constraints. In the continuous perceptron the domain of $\underline{\sigma}$ is \mathbb{R}^N and the constraints impose the positivity of the scalar products (22.1). The instance of the CSP, also called formula in the following, is said satisfiable if there exists a solution (an assignment of $\underline{\sigma}$ fulfilling all the constraints). The k -SAT problem is a boolean CSP ($\mathcal{X} = \{\text{True}, \text{False}\}$) where each constraint (clause) is the disjunction (logical OR) of k literals (a variable or its negation). Similarly in k -XORSAT the literals are combined by an exclusive OR operation, or equivalently an addition modulo 2 of 0/1 boolean variables is required to take a given value. The worst-case complexities of these two problems are very different (k -XORSAT is in the P complexity class for any k while k -SAT is NP-complete for any $k \geq 3$), yet for the issues of this review we shall see that they present a lot of similarities. In the following we use the statistical mechanics convention and represent boolean variables by Ising spins, $\mathcal{X} = \{-1, +1\}$. A k -SAT clause will be defined by k indices $i_1, \dots, i_k \in [1, N]$ and k values $J_{i_1}, \dots, J_{i_k} = \pm 1$, such that the clause is unsatisfied by the assignment $\underline{\sigma}$ if and only if $\sigma_{i_j} = J_{i_j} \quad \forall j \in [1, k]$. A k -XORSAT clause is satisfied if the product of the spins is equal to a fixed value, $\sigma_{i_1} \dots \sigma_{i_k} = J$.

- random Constraint Satisfaction Problem (rCSP)

The set of instances of most CSP can be turned in a probability space by defining a distribution over the sets of instances, as was done in the per-

ceptron case by drawing the vertices \underline{T}^a uniformly on the hypersphere. The random k -SAT formulas considered in the following are obtained by choosing for each clause a independently a k -uplet of distinct indices i_1^a, \dots, i_k^a uniformly over the $\binom{N}{k}$ possible ones, and negating or not the corresponding literals ($J_i^a = \pm 1$) with equal probability one-half. The indices of random XORSAT formulas are chosen similarly, with the constant $J^a = \pm 1$ uniformly.

- thermodynamic limit and phase transitions

These two terms are the physics jargon for, respectively, the large size limit ($N \rightarrow \infty$) and for threshold phenomena as stated for instance in (22.3). In the thermodynamic limit the typical behavior of physical systems is controlled by a small number of parameters, for instance the temperature and pressure of a gas. At a phase transition these systems are drastically altered by a tiny change of a control parameter, think for instance at what happens to water when its temperature crosses 100°C . This critical value of the temperature separates two qualitatively distinct phases, liquid and gaseous. For random CSPs the role of control parameter is usually played by the ratio of constraints per variable, $\alpha = M/N$, kept constant in the thermodynamic limit. Eq. (22.3) describes a satisfiability transition for the continuous perceptron, the critical value $\alpha_s = 2$ separating a satisfiable phase at low α where instances typically have solutions to a phase where they typically do not. Typically is used here as a synonym for with high probability, i.e. with a probability which goes to one in the thermodynamic limit.

- Finite Size Scaling (FSS)

The refined description of the neighborhood of the critical value of α provided by (22.4) is known as a finite size scaling relation. More generally the finite size scaling hypothesis for a threshold phenomenon takes the form

$$\lim_{N \rightarrow \infty} P(N, M = N\alpha_s(1 + \lambda N^{-1/\nu})) = \mathcal{F}(\lambda), \quad (22.5)$$

where ν is called the FSS exponent (2 for the continuous perceptron) and the scaling function $\mathcal{F}(\lambda)$ has limits 1 and 0 at respectively $-\infty$ and $+\infty$. This means that, for a large but finite size N , the transition window for the values of M/N where the probability drops from $1 - \epsilon$ down to ϵ is, for arbitrary small ϵ , of width $N^{-1/\nu}$. Results of this flavour are familiar in the study of random graphs [JLR00]; for instance the appearance of a giant component containing a finite fraction of the vertices of an Erdős-Rényi random graph happens on a window of width $N^{-1/3}$ on the average connectivity. FSS relations are important, not only from the theoretical point of view, but also for practical applications. Indeed numerical experiments are always performed on finite-size instances while theoretical predictions on phase transitions are usually true in the $N \rightarrow \infty$ limit. Finite-size scaling relations help to bridge the gap between the two. We shall review some FSS results in Sec. 22.3.5.

Let us emphasize that random k -SAT, and other random CSP, are expected to share some features of the continuous perceptron model, for instance the existence

of a satisfiability threshold, but of course not its extreme analytical simplicity. In fact, despite an intensive research activity, the mere existence of a satisfiability threshold for random SAT formulas remains a (widely accepted) conjecture. A significant achievement towards the resolution of the conjecture was the proof by Friedgut of the existence of a non-uniform sharp threshold [Fri99]. There exists also upper [Dub01] and lower [Fra01] bounds on the possible location of this putative threshold, which become almost tight for large values of k [AP04]. We refer the reader to Part 1, Chapter 10 of this volume for more details on these issues. This difficulty to obtain tight results with the currently available rigorous techniques is a motivation for the use of heuristic statistical mechanics methods, that provide intuitions on why the standard mathematical ones run into trouble and how to amend them. In the recent years important results first conjectured by physicists were indeed rigorously proven. Before describing in some generality the statistical mechanics approach, it is instructive to study a simple variation of the perceptron model for which the basic probabilistic techniques become inefficient.

22.2.3. The perceptron problem continued: binary variables

The binary perceptron problem consists in looking for solutions of (22.1) on the hypercube i.e. the domain of the variable $\underline{\sigma}$ is $\mathcal{X}^N = \{-1, +1\}^N$ instead of \mathbb{R}^N . This decision problem is NP-complete. Unfortunately Cover's calculation [Cov65] cannot be extended to this case, though it is natural to expect a similar satisfiability threshold phenomenon at an a priori distinct value α_s . Let us first try to study this point with basic probabilistic tools, namely the first and second moment method [AS00]. The former is an application of the Markov inequality,

$$\text{Prob}[Z > 0] \leq \mathbb{E}[Z] , \quad (22.6)$$

valid for positive integer valued random variables Z . We shall use it taking for Z the number of solutions of (22.1),

$$Z = \sum_{\underline{\sigma} \in \mathcal{X}^N} \prod_{a=1}^M \theta(\underline{\sigma} \cdot \underline{T}^a) , \quad (22.7)$$

where $\theta(x) = 1$ if $x > 0$, 0 if $x \leq 0$. The expectation of the number of solutions is easily computed,

$$\mathbb{E}[Z] = 2^N \times 2^{-M} = e^{N G_1} \quad \text{with} \quad G_1 = (1 - \alpha) \ln 2 , \quad (22.8)$$

and vanishes when $N \rightarrow \infty$ if $\alpha > 1$. Hence, from Markov's inequality (22.6), with high probability constraints (22.1) have no solution on the hypercube when the ratio α exceeds unity: if the threshold α_s exists, it must satisfy the bound $\alpha_s \leq 1$. One can look for a lower bound to α_s using the second moment method, relying on the inequality [AS00]

$$\frac{\mathbb{E}[Z]^2}{\mathbb{E}[Z^2]} \leq \text{Prob}[Z > 0] . \quad (22.9)$$

The expectation of the squared number of solutions reads

$$\mathbb{E}[Z^2] = \sum_{\underline{\sigma}, \underline{\sigma}'} (\mathbb{E}[\theta(\underline{\sigma} \cdot \underline{T}) \theta(\underline{\sigma}' \cdot \underline{T})])^M \tag{22.10}$$

since the vertices \underline{T}^a are chosen independently of each other. The expectation on the right hand side of the above expression is simply the probability that the vector pointing to a randomly chosen vertex, \underline{T} , has positive scalar product with both vectors $\underline{\sigma}, \underline{\sigma}'$. Elementary geometrical considerations reveal that

$$\mathbb{E}[\theta(\underline{\sigma} \cdot \underline{T}) \theta(\underline{\sigma}' \cdot \underline{T})] = \frac{1}{2\pi} (\pi - \varphi(\underline{\sigma}, \underline{\sigma}')) \tag{22.11}$$

where φ is the relative angle between the two vectors. This angle can be alternatively parametrized by the overlap between $\underline{\sigma}$ and $\underline{\sigma}'$, i.e. the normalized scalar product,

$$q = \frac{1}{N} \sum_{i=1}^N \sigma_i \sigma'_i = 1 - 2 \frac{1}{N} \sum_{i=1}^N \mathbb{I}(\sigma_i \neq \sigma'_i) . \tag{22.12}$$

The last expression, in which $\mathbb{I}(E)$ denotes the indicator function of the event E , reveals the correspondence between the concept of overlap and the more traditional Hamming distance. The sum over vectors in (22.10) can then be replaced by a sum over overlap values with appropriate combinatorial coefficients counting the number of pairs of vectors at a given overlap. The outcome is

$$\mathbb{E}[Z^2] = 2^N \sum_{q=-1, -1+\frac{2}{N}, -1+\frac{4}{N}, \dots, 1} \binom{N}{N \left(\frac{1+q}{2}\right)} \left(\frac{1}{2} - \frac{1}{2\pi} \text{Arcos } q\right)^M . \tag{22.13}$$

The main point in the above equation is that the sum (22.10) with exponentially many terms is organized as a sum over polynomially many equivalence classes (of exponential size each). It is this reorganization of the sum that makes it possible to essentially replace summation with maximization when N gets large. Using the Laplace method,

$$\lim_{N \rightarrow \infty} \frac{1}{N} \ln \mathbb{E}[Z^2] = \max_{-1 < q < 1} G_2(q) , \tag{22.14}$$

where

$$G_2(q) = \ln 2 - \left(\frac{1+q}{2}\right) \ln \left(\frac{1+q}{2}\right) - \left(\frac{1-q}{2}\right) \ln \left(\frac{1-q}{2}\right) + \alpha \ln \left(\frac{1}{2} - \frac{1}{2\pi} \text{Arcos } q\right) . \tag{22.15}$$

Unfortunately no useful lower bound to α_s can be obtained from such a direct application of the second moment method. Indeed, maximization of G_2 (22.15) over q shows that $\mathbb{E}[Z^2] \gg (\mathbb{E}[Z])^2$ when N diverges, whenever $\alpha > 0$, and in consequence the left hand side of (22.9) vanishes. What causes this failure, and how it can be cured with the use of the 'replica method' will be discussed in Section 22.2.5.

Note that the perceptron problem is not as far as it could seem from the main subject of this review. There exists indeed a natural mapping between the binary perceptron problem and k -SAT. Assume the vertices \underline{T} of the perceptron problem, instead of being drawn on the hypersphere, have coordinates that can take three values: $T_i = -1, 0, 1$. Consider now a k -SAT formula F . To each clause a of F we associate the vertex \underline{T}^a with coordinates $T_i^a = -J_i^a$ if variable i appears in clause a , 0 otherwise. Of course $\sum_i |T_i^a| = k$: exactly k coordinates have non zero values for each vertex. Then replace condition (22.1) with

$$\sum_{i=1}^N \sigma_i T_i^a > -(k-1), \quad \forall a = 1, \dots, M. \quad (22.16)$$

The scalar product is not required to be positive any longer, but to be larger than $-(k-1)$. It is an easy check that the perceptron problem admits a solution on the hypercube ($\sigma_i = \pm 1$) if and only if F is satisfiable. While in the binary perceptron model all coordinates are non-vanishing, only a finite number of them take non zero values in k -SAT. For this reason k -SAT is called a diluted model in statistical physics.

22.2.4. From random CSP to statistical mechanics of disordered systems

The binary perceptron example demonstrated that, while the number of solutions Z of a satisfiable random CSP usually scales exponentially with the size of the problem, large fluctuations can prevent the direct use of standard moment methods¹. The concepts and computation techniques used to tackle this difficulty were in fact developed in an apparently different field, the statistical mechanics of disordered systems [MPV87].

Let us review some basic concepts of statistical mechanics (for introductory books see for example [Ma85, Hua90]). A physical system can be modeled by a space of configuration $\underline{\sigma} \in \mathcal{X}^N$, on which an energy function $E(\underline{\sigma})$ is defined. For instance usual magnets are described by Ising spins $\sigma_i = \pm 1$, the energy being minimized when adjacent spins take the same value. The equilibrium properties of a physical system at temperature T are given by the Gibbs-Boltzmann probability measure on \mathcal{X}^N ,

$$\mu(\underline{\sigma}) = \frac{1}{Z} \exp[-\beta E(\underline{\sigma})], \quad (22.17)$$

where the inverse temperature β equals $1/T$ and Z is a normalization called partition function. The energy function E has a natural scaling, linear in the number N of variables (such a quantity is said to be extensive). In consequence in the thermodynamic limit the Gibbs-Boltzmann measure concentrates on configurations with a given energy density ($e = E/N$), which depends on the conjugated parameter β . The number of such configurations is usually exponentially large, $\approx \exp[Ns]$, with s called the entropy density. The partition function is thus dominated by the contribution of these configurations, hence $\lim(\ln Z/N) = s - \beta e$.

¹The second moment method fails for the random k -SAT problem; yet a refined version of it was used in [AP04], which leads to asymptotically (at large k) tight bounds on the location of the satisfiability threshold.

In the above presentation we supposed the energy to be a simple, known function of the configurations. In fact some magnetic compounds, called spin-glasses, are intrinsically disordered on a microscopic scale. This means that there is no hope in describing exactly their microscopic details, but that one should rather assume their energy to be itself a random function with a known distribution. Hopefully in the thermodynamic limit the fluctuations of the thermodynamic observables as the energy and entropy density vanish, hence the properties of a typical sample will be closely described by the average (over the distribution of the energy function) of the entropy and energy density.

The random CSPs fit naturally in this line of research. The energy function $E(\underline{\sigma})$ of a CSP is defined as the number of constraints violated by the assignment $\underline{\sigma}$, in other words this is the cost function to be minimized in the associated optimization problem (MAXSAT for instance). Moreover the distribution of random instances of CSP is the counterpart of the distribution over the microscopic description of a disordered solid. The study of the optimal configurations of a CSP, and in particular the characterization of a satisfiability phase transition, is achieved by taking the $\beta \rightarrow \infty$ limit. Indeed, when this parameter increases (or equivalently the temperature goes to 0), the law (22.17) favors the lowest energy configurations. In particular if the formula is satisfiable μ becomes the uniform measure over the solutions. Two important features of the formula can be deduced from the behavior of Z at large β : the ground-state energy $E_g = \min_{\underline{\sigma}} E(\underline{\sigma})$, which indicates how good are the optimal configurations, and the ground state entropy $S_g = \ln(|\{\underline{\sigma} : E(\underline{\sigma}) = E_g\}|)$, which counts the number of these optimal configurations. The satisfiability of a formula is equivalent to its ground-state energy being equal to 0. In the large N limit these two thermodynamic quantities are supposed to concentrate around their mean values (this is proven for E in [BFU93]), we thus introduce the associated typical densities,

$$e_g(\alpha) = \lim_{N \rightarrow \infty} \frac{1}{N} \mathbb{E}[E_g], \quad s_g(\alpha) = \lim_{N \rightarrow \infty} \frac{1}{N} \mathbb{E}[S_g]. \quad (22.18)$$

Notice that, in the satisfiable phase, s_g is simply the expected log of the number of solutions.

Some criteria are needed to relate these thermodynamic quantities to the (presumed) satisfiability threshold α_s . A first approach, used for instance in [MZ97], consists in locating it as the point where the ground-state energy density e_g becomes positive. The assumption underlying this reasoning is the absence of an intermediate, typically UNSAT regime, with a sub-extensive positive E_g . In the discussion of the binary perceptron we used another criterion, namely we recognized α_s as the ratio at which the ground-state entropy density vanishes. This argument will be true if the typical number of solutions vanishes continuously at α_s . It is easy to realize that this is not the case for random k -SAT: at any finite value of α a finite fraction $\exp[-\alpha k]$ of the variables do not appear in any clause, which leads to a trivial lower bound $(\ln 2) \exp[-\alpha k]$ on s_g (in the satisfiable regime). This quantity is thus finite at the transition, a large number of solutions disappear suddenly at α_s . Even if it is wrong, the criterion $s_g(\alpha) = 0$ for the determination of the satisfiability transition is instructive for two reasons. First, it becomes asymptotically correct at large k (free variables are very rare

in this limit), this is why it works for the binary perceptron of Section 22.2.3 (which is, as we have seen, close to k -SAT with k of order N). Second, it will reappear below in a refined version: we shall indeed decompose the entropy in two qualitatively distinct contributions, one of the two being indeed vanishing at the satisfiability transition.

22.2.5. Large deviations and the replica method

We have seen in Section 22.2.3 that the number of solutions of the binary perceptron exhibits large fluctuations around its expectation *i.e.* $\mathbb{E}[Z^2] \gg (\mathbb{E}[Z])^2$ when N diverges, whenever $\alpha > 0$. A possible scenario which explains this absence of concentration of the number of solutions is the following. As shown by the moment calculation the natural scaling of Z is exponentially large in N (as is the total configuration space \mathcal{X}^N). We shall thus denote $s = (\ln Z)/N$ the random variable counting the log of the number of solutions. Suppose s follows a large deviation principle [DZ98] that we state in a very rough way as $\text{Prob}[s] \approx \exp[NL(s)]$, with $L(s)$ a negative rate function, assumed for simplicity to be concave. Then the moments of Z are given, at the leading exponential order, by

$$\lim_{N \rightarrow \infty} \frac{1}{N} \ln \mathbb{E}[Z^n] = \max_s [L(s) + ns], \quad (22.19)$$

and are controlled by the values of s such that $L'(s) = -n$. The moments of larger and larger order n are thus dominated by the contribution of rarer and rarer instances with larger and larger numbers of solutions. On the contrary the typical value of the number of solutions is given by the maximum of L , reached in a value we denote $s_g(\alpha)$: with high probability when $N \rightarrow \infty$, Z is comprised between $e^{N(s_g(\alpha) - \epsilon)}$ and $e^{N(s_g(\alpha) + \epsilon)}$, for any $\epsilon > 0$. From this reasoning it appears that the relevant quantity to be computed is

$$s_g(\alpha) = \lim_{N \rightarrow \infty} \frac{1}{N} \mathbb{E}[\ln Z] = \lim_{N \rightarrow \infty} \lim_{n \rightarrow 0} \frac{1}{nN} \ln \mathbb{E}[Z^n]. \quad (22.20)$$

This idea of computing moments of vanishing order is known in statistical mechanics as the replica² method [MPV87]. Its non-rigorous implementation consists in determining the moments of integer order n , which are then continued towards $n = 0$. The outcome of such a computation for the binary perceptron problem reads [KM89]

$$s_g(\alpha) = \max_{q, \hat{q}} \left\{ -\frac{1}{2}q(1 - \hat{q}) + \int_{-\infty}^{\infty} Dz \ln(2 \cosh(z\sqrt{\hat{q}})) + \alpha \int_{-\infty}^{\infty} Dz \ln \left[\int_{z\sqrt{q/(1-q)}}^{\infty} Dy \right] \right\}, \quad (22.21)$$

where $Dz \equiv dz e^{-z^2/2}/\sqrt{2\pi}$. The entropy $s_g(\alpha)$ is a decreasing function of α , which vanishes in $\alpha_s \simeq 0.833$. Numerical experiments support this value for the critical ratio of the satisfiable/unsatisfiable phase transition.

²The vocable replicas comes from the presence of n copies of the vector $\underline{\sigma}$ in the calculation of Z^n (see the $n = 2$ case in formula (22.10)).

The calculation of the second moment is naturally related to the determination of the value of the overlap q between pairs of solutions (or equivalently their Hamming distance, recall Eq. (22.12)). This conclusion extends to the calculation of the n^{th} moment for any integer n , and to the $n \rightarrow 0$ limit. The value of q maximizing the r.h.s. of (22.21), $q^*(\alpha)$, represents the average overlap between two solutions of the same set of constraints (22.1). Actually the distribution of overlaps is highly concentrated in the large N limit around $q^*(\alpha)$, in other words the (reduced) Hamming distance between two solutions is, with high probability, equal to $d^*(\alpha) = (1 - q^*(\alpha))/2$. This distance $d^*(\alpha)$ ranges from $\frac{1}{2}$ for $\alpha = 0$ to $\simeq \frac{1}{4}$ at $\alpha = \alpha_s$. Slightly below the critical ratio solutions are still far away from each other on the hypercube³.

22.3. Phase transitions in random CSPs

22.3.1. The clustering phenomenon

We have seen that the statistical physics approach to the perceptron problem naturally provided us with information about the geometry of the space of its solutions. Maybe one of the most important contribution of physicists to the field of random CSP was to suggest the presence of further phase transitions in the satisfiable regime $\alpha < \alpha_s$, affecting qualitatively the geometry (structure) of the set of solutions [BMW00, MZ02, KMRT⁺07].

This subset of the configuration space is indeed thought to break down into “clusters” in a part of the satisfiable phase, $\alpha \in [\alpha_d, \alpha_s]$, α_d being the threshold value for the clustering transition. Clusters are meant as a partition of the set of solutions having certain properties listed below. Each cluster contains an exponential number of solutions, $\exp[Ns_{\text{int}}]$, and the clusters are themselves exponentially numerous, $\exp[N\Sigma]$. The total entropy density thus decomposes into the sum of s_{int} , the internal entropy of the clusters and Σ , encoding the number of these clusters, usually termed complexity in this context. Furthermore, solutions inside a given cluster should be well-connected, while two solutions of distinct clusters are well-separated. A possible definition for these notions is the following. Suppose $\underline{\sigma}$ and $\underline{\tau}$ are two solutions of a given cluster. Then one can construct a path ($\underline{\sigma} = \underline{\sigma}_0, \underline{\sigma}_1, \dots, \underline{\sigma}_{n-1}, \underline{\sigma}_n = \underline{\tau}$) where any two successive $\underline{\sigma}_i$ are separated by a sub-extensive Hamming distance. On the contrary such a path does not exist if $\underline{\sigma}$ and $\underline{\tau}$ belong to two distinct clusters. Clustered configuration spaces as described above have been often encountered in various contexts, e.g. neural networks [MO94] and mean-field spin glasses [KT87]. A vast body of involved, yet non-rigorous, analytical techniques [MPV87] have been developed in the field of statistical mechanics of disordered systems to tackle such situations, some of them having been justified rigorously [Tal03, PT04, FL03]. In this literature clusters appear under the name of “pure states”, or “lumps” (see for instance the chapter 6 of [Tal03] for a rigorous definition and proof of existence in a related model). As we shall explain in a few lines, this clustering phenomenon has been demonstrated rigorously in the case of random XORSAT

³This situation is very different from the continuous perceptron case, where the typical overlap $q^*(\alpha)$ reaches one when α tends to 2: a single solution is left right at the critical ratio.

instances [MRTZ03, CDMM03]. For random SAT instances, where in fact the detailed picture of the satisfiable phase is thought to be richer [KMRT⁺07], there are some rigorous results [MMZ05, DMMZ08, ART06] on the existence of clusters for $k \geq 8$.

22.3.2. Phase transitions in random XORSAT

Consider an instance F of the XORSAT problem [RTWZ01], i.e. a list of M linear equations each involving k out of N boolean variables, where the additions are computed modulo 2. The study performed in [MRTZ03, CDMM03] provides a detailed picture of the clustering and satisfiability transition sketched above. A crucial point is the construction of a core subformula according to the following algorithm. Let us denote $F_0 = F$ the initial set of equations, and V_0 the set of variables which appear in at least one equation of F_0 . A sequence F_T, V_T is constructed recursively: if there are no variables in V_T which appear in exactly one equation of F_T the algorithm stops. Otherwise one of these “leaf variables” σ_i is chosen arbitrarily, F_{T+1} is constructed from F_T by removing the unique equation in which σ_i appeared, and V_{T+1} is defined as the set of variables which appear at least once in F_{T+1} . Let us call T_* the number of steps performed before the algorithm stops, and $F' = F_{T_*}$, $V' = V_{T_*}$ the remaining clauses and variables. Note first that despite the arbitrariness in the choice of the removed leaves, the output subformula F' is unambiguously determined by F . Indeed, F' can be defined as the maximal (in the inclusion sense) subformula in which all present variables have a minimal occurrence number of 2, and is thus unique. In graph theoretic terminology F' is the 2-core of F , the q -core of hypergraphs being a generalization of the more familiar notion on graphs, thoroughly studied in random graph ensembles in [PSW96]. Extending this study, relying on the approximability of this leaf removal process by differential equations [Kur70], it was shown in [MRTZ03, CDMM03] that there is a threshold phenomenon at $\alpha_d(k)$. For $\alpha < \alpha_d$ the 2-core F' is, with high probability, empty, whereas it contains a finite fraction of the variables and equations for $\alpha > \alpha_d$. α_d is easily determined numerically: it is the smallest value of α such that the equation $x = 1 - \exp[-\alpha k x^{k-1}]$ has a non-trivial solution in $(0, 1]$.

It turns out that F is satisfiable if and only if F' is, and that the number of solutions of these two formulas are related in an enlightening way. It is clear that if the 2-core has no solution, there is no way to find one for the full formula. Suppose on the contrary that an assignment of the variables in V' that satisfy the equations of F' has been found, and let us show how to construct a solution of F (and count in how many possible ways we can do this). Set $\mathcal{N}_0 = 1$, and reintroduce step by step the removed equations, starting from the last: in the n 'th step of this new procedure we reintroduce the clause which was removed at step $T_* - n$ of the leaf removal. This reintroduced clause has $d_n = |V_{T_* - n - 1}| - |V_{T_* - n}| \geq 1$ leaves; their configuration can be chosen in $2^{d_n - 1}$ ways to satisfy the reintroduced clause, irrespectively of the previous choices, and we bookkeep this number of possible extensions by setting $\mathcal{N}_{n+1} = \mathcal{N}_n 2^{d_n - 1}$. Finally the total number of solutions of F compatible with the choice of the solution of F' is obtained by adding the freedom of the variables which appeared in no equations of F , $\mathcal{N}_{\text{int}} = \mathcal{N}_{T_*} 2^{N - |V_0|}$.

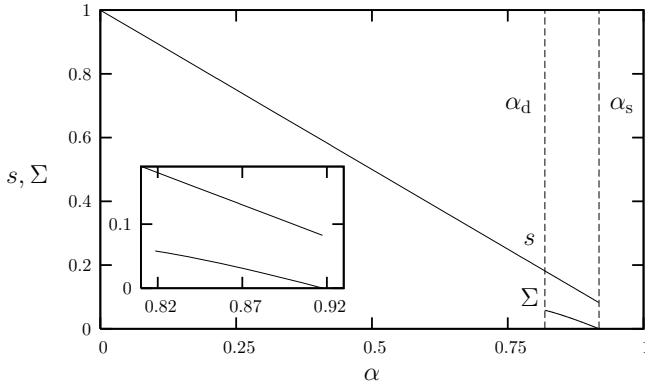


Figure 22.2. Complexity and total entropy for 3-XORSAT, in units of $\ln 2$. The inset presents an enlargement of the regime $\alpha \in [\alpha_d, \alpha_s]$.

Let us underline that \mathcal{N}_{int} is independent of the initial satisfying assignment of the variables in V' , as appears clearly from the description of the reconstruction algorithm; this property can be traced back to the linear algebra structure of the problem. This suggests naturally the decomposition of the total number of solutions of F as the product of the number of satisfying assignments of V' , call it $\mathcal{N}_{\text{core}}$, by the number of compatible full solutions \mathcal{N}_{int} . In terms of the associated entropy densities this decomposition is additive

$$s = \Sigma + s_{\text{int}}, \quad \Sigma \equiv \frac{1}{N} \ln \mathcal{N}_{\text{core}}, \quad s_{\text{int}} \equiv \frac{1}{N} \ln \mathcal{N}_{\text{int}}, \quad (22.22)$$

where the quantity Σ is the entropy density associated to the core of the formula. It is in fact much easier technically to compute the statistical (with respect to the choice of the random formula F) properties of Σ and s_{int} once this decomposition has been done (the fluctuations in the number of solutions is much smaller once the non-core part of the formula has been removed, and the second moment method can be applied). The outcome of the computations [MRTZ03, CDMM03] is the determination of the threshold value α_s for the appearance of a solution of the 2-core F' (and thus of the complete formula), along with explicit formulas for the typical values of Σ and s . These two quantities are plotted on Fig. 22.2. The satisfiability threshold corresponds to the cancellation of Σ : the number of solutions of the core vanishes continuously at α_s , while the total entropy remains finite because of the freedom of choice for the variables in the non-core part of the formula.

On top of the simplification in the analytical determination of the satisfiability threshold, this core decomposition of a formula unveils the change in the structure of the set of solutions that occurs at α_d . Indeed, let us call clusters the sets of solutions of F reconstructed from a common solution of F' . Then one can show that this partition of the solution set of F exhibits the properties exposed in Sec. 22.3.1, namely that solutions are well-connected inside a cluster and separated from one cluster to another. The number of clusters is precisely equal to the number of solutions of the core subformula, it thus undergoes a drastic change at

Table 22.1. Critical connectivities for the dynamical, condensation and satisfiability transitions for k -SAT random formulas.

	α_d [KMRT ⁺ 07]	α_c [KMRT ⁺ 07]	α_s [MMZ06]
$k = 3$	3.86	3.86	4.267
$k = 4$	9.38	9.547	9.93
$k = 5$	19.16	20.80	21.12
$k = 6$	36.53	43.08	43.4

α_d . For smaller ratio of constraints the core is typically empty, there is one single cluster containing all solutions; when the threshold α_d is reached there appears an exponential numbers of clusters, the rate of growth of this exponential being given by the complexity Σ . Before considering the extension of this picture to random SAT problems, let us mention that further studies of the geometry of the space of solutions of random XORSAT instances can be found in [MS06a, MM06].

22.3.3. Phase transitions in random SAT

The possibility of a clustering transition in random SAT problems was first studied in [BMW00] by means of variational approximations. Later developments allowed the computation of the complexity and, from the condition of its vanishing, the estimation of the satisfiability threshold α_s . This was first done for $k = 3$ in [MZ02] and generalized for $k \geq 4$ in [MMZ06], some of the values of α_s thus computed are reported in Tab. 22.1. A systematic expansion of α_s at large k was also performed in [MMZ06].

SAT formulas do not share the linear algebra structure of XORSAT, which makes the analysis of the clustering transition much more difficult, and leads to a richer structure of the satisfiable phase $\alpha \leq \alpha_s$. The simple arguments are not valid anymore, one cannot extract a core subformula from which the partition of the solutions into clusters follows directly. It is thus necessary to define them as a partition of the solutions such that each cluster is well-connected and well-separated from the other ones. A second complication arises: there is no reason for the clusters to contain all the same number of solutions, as was ensured by the linear structure of XORSAT. On the contrary, as was observed in [BMW00] and in [MPR05] for the similar random COL problem, one faces a variety of clusters with various internal entropies s_{int} . The complexity Σ becomes a function of s_{int} , in other words the number of clusters of internal entropy density s_{int} is typically exponential, growing at the leading order like $\exp[N\Sigma(s_{\text{int}})]$. Drawing the consequences of these observations, a refined picture of the satisfiable phase, and in particular the existence of a new (so-called condensation) threshold $\alpha_c \in [\alpha_d, \alpha_s]$, was advocated in [KMRT⁺07]. Let us briefly sketch some of these new features and their relationship with the previous results of [MZ02, MMZ06]. Assuming the existence of a positive, concave, complexity function $\Sigma(s_{\text{int}})$, continuously vanishing outside an interval of internal entropy densities $[s_-, s_+]$, the total entropy density is given by

$$s = \lim_{N \rightarrow \infty} \frac{1}{N} \ln \int_{s_-}^{s_+} ds_{\text{int}} e^{N[\Sigma(s_{\text{int}}) + s_{\text{int}}]} . \quad (22.23)$$

In the thermodynamic limit the integral can be evaluated with the Laplace method. Two qualitatively distinct situations can arise, whether the integral is dominated by a critical point in the interior of the interval $[s_-, s_+]$, or by the neighborhood of the upper limit s_+ . In the former case an overwhelming majority of the solutions are contained in an exponential number of clusters, while in the latter the dominant contributions comes from a sub-exponential number of clusters of internal entropy s_+ , as $\Sigma(s_+) = 0$. The threshold α_c separates the first regime $[\alpha_d, \alpha_c]$ where the relevant clusters are exponentially numerous, from the second, condensated situation for $\alpha \in [\alpha_c, \alpha_s]$ with a sub-exponential number of dominant clusters⁴.

The computations of [MZ02, MMZ06] did not take into account the distribution of the various internal entropies of the clusters, which explains the discrepancy in the estimation of the clustering threshold α_d between [MZ02, MMZ06] and [KMRT⁺07]. Let us however emphasize that this refinement of the picture does not contradict the estimation of the satisfiability threshold of [MZ02, MMZ06]: the complexity computed in these works is Σ_{\max} , the maximal value of $\Sigma(s_{\text{int}})$ reached at a local maximum with $\Sigma'(s) = 0$, which indeed vanishes when the whole complexity function disappears.

It is fair to say that the details of the picture proposed by statistical mechanics studies have rapidly evolved in the last years, and might still be improved. They rely indeed on self-consistent assumptions which are rather tedious to check [MPRT04]. Some elements of the clustering scenario have however been established rigorously in [MMZ05] (see also [DMMZ08, ART06] for more results). In particular, for some values of (large enough) k and α in the satisfiable regime, there exist forbidden intermediate Hamming distances between pairs of configurations, which are either close (in the same cluster) or far apart (in two distinct clusters).

Note finally that the consequences of such distributions of clusters internal entropies were investigated on a toy model in [MZ08], and that yet another threshold $\alpha_f > \alpha_d$ for the appearance of frozen variables constrained to take the same values in all solutions of a given cluster was investigated in [Sem08].

22.3.4. A glimpse at the computations

The statistical mechanics of disordered systems [MPV87] was first developed on so-called fully-connected models, where each variable appears in a number of constraints which diverges in the thermodynamic limit. This is for instance the case of the perceptron problem discussed in Sec. 22.2. On the contrary, in a random k -SAT instance a variable is typically involved in a finite number of clauses, one speaks in this case of a diluted model. This finite connectivity is a source of major technical complications. In particular the replica method, alluded to in Sec. 22.2.3 and applied to random k -SAT in [MZ97, BMW00], turns out to be rather cumbersome for diluted models in the presence of clustering [Mon98]. The cavity formalism [MP01, MP03, MZ02], formally equivalent to the replica one, is more adapted to the diluted models. In the following paragraphs we shall

⁴This picture is expected to hold for $k \geq 4$; for $k = 3$, the dominant clusters are expected to be of sub-exponential number in the whole clustered phase, hence $\alpha_c = \alpha_d$ in this case.

try to give a few hints at the strategy underlying the cavity computations, that might hopefully ease the reading of the original literature.

The description of the random formula ensemble has two complementary aspects: a global (thermodynamic) one, which amounts to the computation of the typical energy and number of optimal configurations. A more ambitious description will also provide geometrical information on the organization of this set of optimal configurations inside the N -dimensional hypercube. As discussed above these two aspects are in fact interleaved, the clustering affecting both the thermodynamics (by the decomposition of the entropy into the complexity and the internal entropy) and the geometry of the configuration space. Let us for simplicity concentrate on the $\alpha < \alpha_s$ regime and consider a satisfiable formula F . Both thermodynamic and geometric aspects can be studied in terms of the uniform probability law over the solutions of F :

$$\mu(\underline{\sigma}) = \frac{1}{Z} \prod_{a=1}^M w_a(\underline{\sigma}_a), \quad (22.24)$$

where Z is the number of solutions of F , the product runs over its clauses, and w_a is the indicator function of the event “clause a is satisfied by the assignment $\underline{\sigma}$ ” (in fact this depends only on the configuration of the k variables involved in the clause a , that we denote $\underline{\sigma}_a$). For instance the (information theoretic) entropy of μ is equal to $\ln Z$, the log of the number of solutions, and geometric properties can be studied by computing averages with respect to μ of well-chosen functions of $\underline{\sigma}$.

A convenient representation of such a law are factor graphs [KFL01]. These are bipartite graphs with two types of vertices (see Fig. 22.3 for an illustration): one variable node (filled circle) is associated to each of the N Boolean variables, while the clauses are represented by M constraint nodes (empty squares). By convention we use the indices a, b, \dots for the constraint nodes, i, j, \dots for the variables. An edge is drawn between variable node i and constraint node a if and only if a depends on i . To indicate which value of σ_i satisfies the clause a one can use two type of linestyles, solid and dashed on the figure. A notation repeatedly used in the following is ∂a (resp. ∂i) for the neighborhood of a constraint (resp. variable) node, i.e. the set of adjacent variable (resp. constraint) nodes. In this context \setminus denotes the subtraction from a set. We shall more precisely denote $\partial_+ i(a)$ (resp. $\partial_- i(a)$) the set of clauses in $\partial i \setminus a$ agreeing (resp. disagreeing) with a on the satisfying value of σ_i , and $\partial_\sigma i$ the set of clauses in ∂i which are satisfied by $\sigma_i = \sigma$. This graphical representation naturally suggests a notion of distance between variable nodes i and j , defined as the minimal number of constraint nodes crossed on a path of the factor graph linking nodes i and j .

Suppose now that F is drawn from the random ensemble. The corresponding random factor graph enjoys several interesting properties [JLR00]. The degree $|\partial i|$ of a randomly chosen variable i is, in the thermodynamic limit, a Poisson random variable of average αk . If instead of a node one chooses randomly an edge $a-i$, the outdegree $|\partial i \setminus a|$ of i has again a Poisson distribution with the same parameter. Moreover the sign of the literals being chosen uniformly, independently of the topology of the factor graph, the degrees $|\partial_+ i|$, $|\partial_- i|$, $|\partial_+ i(a)|$ and $|\partial_- i(a)|$ are Poisson random variables of parameter $\alpha k/2$. Another important feature of these

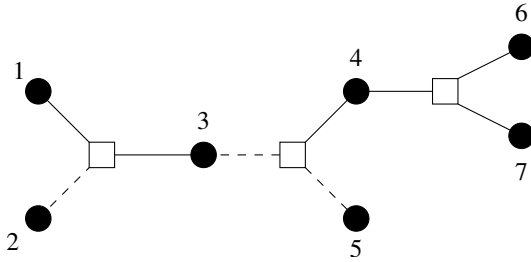


Figure 22.3. The factor graph representation of a small 3-SAT formula: $(x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_3} \vee x_4 \vee \overline{x_5}) \wedge (x_4 \vee x_6 \vee x_7)$.

random factor graphs is their local tree-like character: if the portion of the formula at graph distance smaller than L of a randomly chosen variable is exposed, the probability that this subgraph is a tree goes to 1 if L is kept fixed while the size N goes to infinity.

Let us for a second forget about the rest of the graph and consider a finite formula whose factor graph is a tree, as is the case for the example of Fig. 22.3. The probability law μ of Eq. (22.24) becomes in this case a rather simple object. Tree structures are indeed naturally amenable to a recursive (dynamic programming) treatment, operating first on sub-trees which are then glued together. More precisely, for each edge between a variable node i and a constraint node a one defines the amputated tree $F_{a \rightarrow i}$ (resp. $F_{i \rightarrow a}$) by removing all clauses in ∂i apart from a (resp. removing only a). These subtrees are associated to probability laws $\mu_{a \rightarrow i}$ (resp. $\mu_{i \rightarrow a}$), defined as in Eq. (22.24) but with a product running only on the clauses present in $F_{a \rightarrow i}$ (resp. $F_{i \rightarrow a}$). The marginal law of the root variable i in these amputated probability measures can be parametrized by a single real, as σ_i can take only two values (that, in the Ising spin convention, are ± 1). We thus define these fields, or messages, $h_{i \rightarrow a}$ and $u_{a \rightarrow i}$, by

$$\mu_{i \rightarrow a}(\sigma_i) = \frac{1 - J_i^a \sigma_i \tanh h_{i \rightarrow a}}{2}, \quad \mu_{a \rightarrow i}(\sigma_i) = \frac{1 - J_i^a \sigma_i \tanh u_{a \rightarrow i}}{2}, \quad (22.25)$$

where we recall that $\sigma_i = J_i^a$ is the value of the literal i unsatisfying clause a . A standard reasoning (see for instance [BMZ05]) allows to derive recursive equations (illustrated in Fig. 22.4) on these messages,

$$h_{i \rightarrow a} = \sum_{b \in \partial_+ i(a)} u_{b \rightarrow i} - \sum_{b \in \partial_- i(a)} u_{b \rightarrow i}, \quad (22.26)$$

$$u_{a \rightarrow i} = -\frac{1}{2} \ln \left(1 - \prod_{j \in \partial a \setminus i} \frac{1 - \tanh h_{j \rightarrow a}}{2} \right).$$

Because the factor graph is a tree this set of equations has a unique solution which can be efficiently determined: one start from the leaves (degree 1 variable nodes) which obey the boundary condition $h_{i \rightarrow a} = 0$, and progresses inwards the graph. The law μ can be completely described from the values of the h 's and u 's

solutions of these equations for all edges of the graph. For instance the marginal probability of σ_i can be written as

$$\mu(\sigma_i) = \frac{1 + \sigma_i \tanh h_i}{2}, \quad h_i = \sum_{a \in \partial_+ i} u_{a \rightarrow i} - \sum_{a \in \partial_- i} u_{a \rightarrow i}. \quad (22.27)$$

In addition the entropy s of solutions of such a tree formula, can be computed from the values of the messages h and u [BMZ05].

We shall come back to the equations (22.26), and justify the denomination messages, in Sec. 22.5.3; these can be interpreted as the Belief Propagation [KFL01, YFW01, YFW03] heuristic equations for loopy factor graphs.

The factor graph of random formulas is only locally tree-like; the simple computation sketched above has thus to be amended in order to take into account the effect of the distant, loopy part of the formula. Let us call F_L the factor graph made of variable nodes at graph distance smaller than or equal to L from an arbitrarily chosen variable node i in a large random formula F , and B_L the variable nodes at distance exactly L from i . Without loss of generality in the thermodynamic limit, we can assume that F_L is a tree. The cavity method amounts to an hypothesis on the effect of the distant part of the factor graph, $F \setminus F_L$, i.e. on the boundary condition it induces on F_L . In its simplest (so called replica symmetric) version, that is believed to correctly describe the unclustered situation for $\alpha \leq \alpha_d$, $F \setminus F_L$ is replaced, for each variable node j in the boundary B_L , by a fictitious constraint node which sends a bias $u_{\text{ext} \rightarrow j}$. In other words the boundary condition is factorized on the various nodes of B_L . The factorization property is intuitively sound because, in the amputated factor graph $F \setminus F_L$, the distance between the variables of B_L is typically large (of order $\ln N$), and these variables should thus be weakly correlated. Actually this property holds only when $\alpha \leq \alpha_d$, as strong correlations arise in the clustered phase. The external biases are then turned into random variables to take into account the randomness in the construction of the factor graphs, and Eq. (22.26) acquires a distributional meaning. The messages h (resp. u) are supposed to be i.i.d. random variables drawn from a common distribution, the degrees $\partial_{\pm} i(a)$ being two independent Poisson random variables of parameter $\alpha k/2$. These distributional equations can be numerically solved by a population dynamics algorithm [MP01], also known as a particle representation in the statistics literature. The typical entropy density is then computed by averaging s over these distributions of h and u .

This description fails in the presence of clustering, which induces correlations between the variable nodes of B_L in the amputated factor graph $F \setminus F_L$. To take these correlations into account a refined version of the cavity method (termed one step of replica symmetry breaking, in short 1RSB) has been developed. It relies on the hypothesis that the partition of the solution space into clusters γ has nice decorrelation properties: once decomposed onto this partition, μ restricted to a cluster γ behaves essentially as in the unclustered phase (it is a pure state in statistical mechanics jargon). Each directed edge $a \rightarrow i$ should thus bear a family of messages $u_{a \rightarrow i}^{\gamma}$, one for each cluster, or alternatively a distribution $Q_{a \rightarrow i}(u)$ of the messages with respect to the choice of γ . The equations (22.26) are thus promoted to recursions between distributions $P_{i \rightarrow a}(h)$, $Q_{a \rightarrow i}(u)$, which depends on a real m known as the Parisi breaking parameter. Its role is to select the

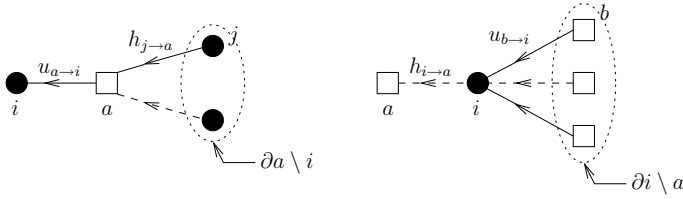


Figure 22.4. A schematic representation of Eq. (22.26).

size of the investigated clusters, i.e. the number of solutions they contain. The computation of the typical entropy density is indeed replaced by a more detailed thermodynamic potential [Mon95],

$$\Phi(m) = \frac{1}{N} \ln \sum_{\gamma} Z_{\gamma}^m = \frac{1}{N} \ln \int_{s_{-}}^{s_{+}} ds_{\text{int}} e^{N[\Sigma(s_{\text{int}}) + m s_{\text{int}}]} . \quad (22.28)$$

In this formula Z_{γ} denotes the number of solutions inside a cluster γ , and we used the hypothesis that at the leading order the number of clusters with internal entropy density s_{int} is given by $\exp[N\Sigma(s_{\text{int}})]$. The complexity function $\Sigma(s_{\text{int}})$ can thus be obtained from $\Phi(m)$ by an inverse Legendre transform. For generic values of m this approach is computationally very demanding; following the same steps as in the replica symmetric version of the cavity method one faces a distribution (with respect to the topology of the factor graph) of distributions (with respect to the choice of the clusters) of messages. Simplifications however arise for $m = 1$ and $m = 0$ [KMRT⁺07]; the latter case corresponds in fact to the original Survey Propagation approach of [MZ02]. As appears clearly in Eq. (22.28), for this value of m all clusters are treated on an equal footing and the dominant contribution comes from the most numerous clusters, independently of their sizes. Moreover, as we further explain in Sec. 22.5.3, the structure of the equations can be greatly simplified in this case, the distribution over the cluster of fields being parametrized by a single number.

22.3.5. Finite Size Scaling results

As we explained in Sec. 22.2.2 the threshold phenomenon can be more precisely described by finite size scaling relations. Let us mention some FSS results about the transitions we just discussed.

For random 2-SAT, where the satisfiability property is known [dlV01] to exhibit a sharp threshold at $\alpha_s = 1$, the width of the transition window has been determined in [BBC⁺01]. The range of α where the probability of satisfaction drops significantly is of order $N^{-1/3}$, i.e. the exponent ν is equal to 3, as for the random graph percolation. This similarity is not surprising, the proof of [BBC⁺01] relies indeed on a mapping of 2-SAT formulas onto random (directed) graphs.

The clustering transition for XORSAT was first conjectured in [AMRU04] (in the related context of error-correcting codes) then proved in [DM07] to be described by

$$P(N, M = N(\alpha_d + N^{-1/2}\lambda + N^{-2/3}\delta)) = \mathcal{F}(\lambda) + O(N^{-5/26}) , \quad (22.29)$$

where δ is a subleading shift correction that has been explicitly computed, and the scaling function \mathcal{F} is, upto a multiplicative factor on λ , the same error function as in Eq. (22.4).

A general result has been proved in [Wil02] on the width of transition windows. Under rather unrestrictive conditions one can show that $\nu \geq 2$: the transitions cannot be arbitrarily sharp. Roughly speaking the bound is valid when a finite fraction of the clauses are not decisive for the property of the formulas studied, for instance clauses containing a leaf variable are not relevant for the satisfiability of a formula. The number of these irrelevant clauses is of order N and has thus natural fluctuations of order \sqrt{N} ; these fluctuations blur the transition window which cannot be sharper than $N^{-1/2}$.

Several studies (see for instance [KS94, MZK⁺99, RTWZ01]) have attempted to determine the transition window from numeric evaluations of the probability $P(N, \alpha)$, for instance for the satisfiability threshold of random 3-SAT [KS94, MZK⁺99] and XORSAT [RTWZ01]. These studies are necessarily confined to small formula sizes, as the typical computation cost of complete algorithms grows exponentially around the transition. In consequence the asymptotic regime of the transition window, $N^{-1/\nu}$, is often hidden by subleading corrections which are difficult to evaluate, and in [KS94, MZK⁺99] the reported values of ν were found to be in contradiction with the latter derived rigorous bound. This is not an isolated case, numerical studies are often plagued by uncontrolled finite-size effects, as for instance in the bootstrap percolation [GLBD05], a variation of the classical percolation problem.

22.4. Local search algorithms

The rest of this review will be devoted to the study of various SAT-solving algorithms. Algorithms are, to some extent, similar to dynamical processes studied in statistical physics. In this context the focus is however mainly on stochastic processes that respect detailed balance with respect to the Gibbs-Boltzmann measure [Cug03], a condition which is rarely respected by solving algorithms. Physics inspired techniques can yet be useful, and will emerge in three different ways. The random walk algorithms considered in this Section are stochastic processes in the space of configurations (not fulfilling the detailed balance condition), moving by small steps where one or a few variables are modified. Out-of-equilibrium physics (and in particular growth processes) provide an interesting view on classical complete algorithms (DPLL), as shown in Sec. 22.5.2. Finally, the picture of the satisfiable phase put forward in Sec. 22.3 underlies the message-passing procedures discussed in Sec. 22.5.3.

22.4.1. Pure random walk sat, definition and results valid for all instances

Papadimitriou [Pap91] proposed the following algorithm, called Pure Random Walk Sat (PRWSAT) in the following, to solve k -SAT formulas:

1. Choose an initial assignment $\underline{\sigma}(0)$ uniformly at random and set $T = 0$.

2. If $\underline{\sigma}(T)$ is a solution of the formula (i.e. $E(\underline{\sigma}(T)) = 0$), output SOLUTION and stop. If $T = T_{\max}$, a threshold fixed beforehand, output UNDETERMINED and stop.
3. Otherwise, pick uniformly at random a clause among those that are UNSAT in $\underline{\sigma}(T)$; pick uniformly at random one of the k variables of this clause and flip it (reverse its status from True to False and vice-versa) to define the next assignment $\underline{\sigma}(T + 1)$; set $T \rightarrow T + 1$ and go back to step 2.

This defines a stochastic process $\underline{\sigma}(T)$, a biased random walk in the space of configurations. The modification $\underline{\sigma}(T) \rightarrow \underline{\sigma}(T + 1)$ in step 3 makes the selected clause satisfied; however the flip of a variable i can turn previously satisfied clauses into unsatisfied ones (those which were satisfied solely by i in $\underline{\sigma}(T)$).

This algorithm is not complete: if it outputs a solution one is certain that the formula was satisfiable (and the current configuration provides a certificate of it), but if no solution has been found within the T_{\max} allowed steps one cannot be sure that the formula was unsatisfiable. There are however two rigorous results which makes it a probabilistically almost complete algorithm [MR95].

For $k = 2$, it was shown in [Pap91] that PRWSAT finds a solution in a time of order $O(N^2)$ with high probability for all satisfiable instances. Hence, one is almost certain that the formula was unsatisfiable if the output of the algorithm is UNDETERMINED after $T_{\max} = O(N^2)$ steps.

Schöningh [Sch02] proposed the following variation for $k = 3$. If the algorithm fails to find a solution before $T_{\max} = 3N$ steps, instead of stopping and printing UNDETERMINED, it restarts from step 1, with a new random initial condition $\underline{\sigma}(0)$. Schöningh proved that if after R restarts no solution has been found, then the probability that the instance is satisfiable is upper-bounded by $\exp[-R \times (3/4)^N]$ (asymptotically in N). This means that a computational cost of order $(4/3)^N$ allows to reduce the probability of error of the algorithm to arbitrary small values. Note that if the time scaling of this bound is exponential, it is also exponentially smaller than the 2^N cost of an exhaustive enumeration. Improvements on the factor $4/3$ are reported in [BS04].

22.4.2. Typical behavior on random k -SAT instances

The results quoted above are true for any k -SAT instance. An interesting phenomenology arises when one applies the PRWSAT algorithm to instances drawn from the random k -SAT ensemble [SM03, BHW03]. Figure 22.5 displays the temporal evolution of the number of unsatisfied clauses during the execution of the algorithm, for two random 3-SAT instances of constraint ratio $\alpha = 2$ and 3. The two curves are very different: at low values of α the energy decays rather fast towards 0, until a point where the algorithm finds a solution and stops. On the other hand, for larger values of α , the energy first decays towards a strictly positive value, around which it fluctuates for a long time, until a large fluctuation reaches 0, signaling the discovery of a solution. A more detailed study with formulas of increasing sizes reveals that a threshold value $\alpha_{\text{rw}} \approx 2.7$ (for $k = 3$) sharply separates this two dynamical regimes. In fact the fraction of unsatisfied clauses $\varphi = E/M$, expressed in terms of the reduced time $t = T/M$, concentrates in the thermodynamic limit around a deterministic function $\varphi(t)$. For $\alpha < \alpha_{\text{rw}}$

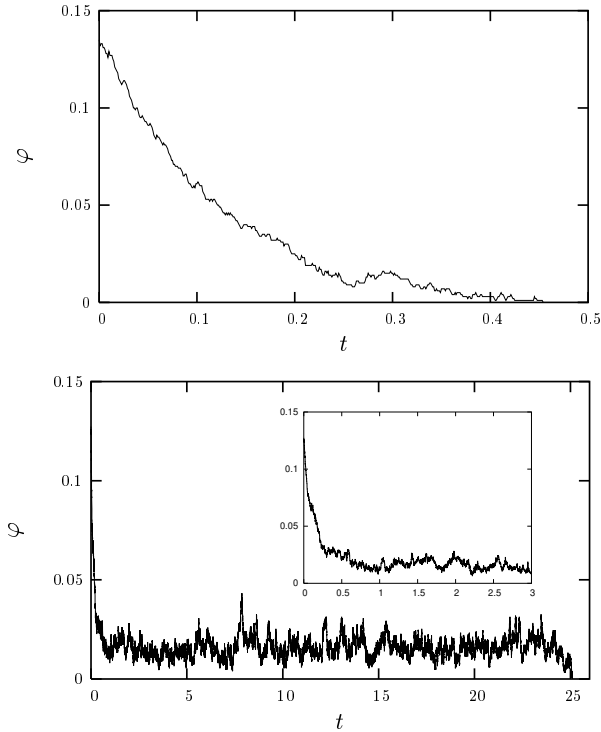


Figure 22.5. Fraction of unsatisfied constraints $\varphi = E/M$ in function of reduced time $t = T/M$ during the execution of PRWSAT on random 3-SAT formulas with $N = 500$ variables. Top: $\alpha = 2$, Bottom: $\alpha = 3$.

the function $\varphi(t)$ reaches 0 at a finite value $t_{\text{sol}}(\alpha, k)$, which means that the algorithm finds a solution in a linear number of steps, typically close to $Nt_{\text{sol}}(\alpha, k)$. On the contrary for $\alpha > \alpha_{\text{rw}}$ the reduced energy $\varphi(t)$ reaches a positive value $\varphi_{\text{as}}(\alpha, k)$ as $t \rightarrow \infty$; a solution, if any, can be found only through large fluctuations of the energy which occur on a time scale exponentially large in N . This is an example of a metastability phenomenon, found in several other stochastic processes, for instance the contact process [Lig85]. When the threshold α_{rw} is reached from below the solving time $t_{\text{sol}}(\alpha, k)$ diverges, while the height of the plateau $\varphi_{\text{as}}(\alpha, k)$ vanishes when α_{rw} is approached from above.

In [SM03, BHW03] various statistical mechanics inspired techniques have been applied to study analytically this phenomenology, some results are presented in Figure 22.6. The low α regime can be tackled by a systematic expansion of $t_{\text{sol}}(\alpha, k)$ in powers of α . The first three terms of these series have been computed, and are shown on the left panel to be in good agreement with the numerical simulations.

Another approach was followed to characterize the transition α_{rw} , and to compute (approximations of) the asymptotic fraction of unsatisfied clauses φ_{as} and the intensity of the fluctuations around it. The idea is to project the Marko-

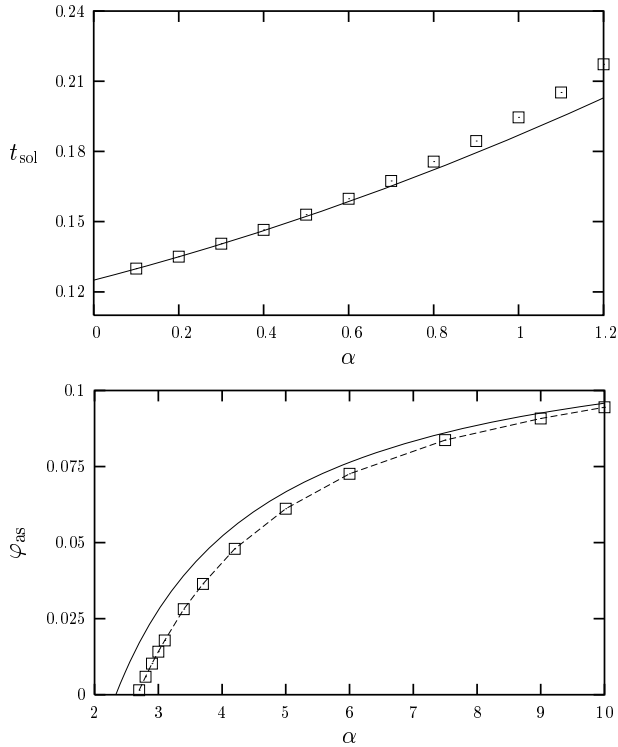


Figure 22.6. Top: linear solving time $t_{\text{sol}}(\alpha, 3)$ for random 3-SAT formulas in function of α ; symbols correspond to numerical simulations, solid line to the second order expansion in α obtained in [SM03]. Bottom: fraction of unsatisfied constraints reached at large time for $\alpha > \alpha_{\text{rw}}$ for random 3-SAT formulas; symbols correspond to numerical simulations, solid line to the approximate analytical computations of [SM03, BHW03].

vian evolution of the configuration $\sigma(T)$ on a simpler observable, the energy $E(T)$. Obviously the Markovian property is lost in this transformation, and the dynamics of $E(T)$ is much more complex. One can however approximate it by assuming that all configurations of the same energy $E(T)$ are equiprobable at a given step of execution of the algorithm. This rough approximation of the evolution of $E(T)$ is found to concentrate around its mean value in the thermodynamic limit, as was observed numerically for the original process. Standard techniques allow to compute this average approximated evolution, which exhibits the threshold behavior explained above at a value $\alpha = (2^k - 1)/k$ which is, for $k = 3$, slightly lower than the threshold α_{rw} . The right panel of Fig. 22.6 confronts the results of this approximation with the numerical simulations; given the roughness of the hypothesis the agreement is rather satisfying, and is expected to improve for larger values of k .

The rigorous results on the behavior of PRWSAT on random instances are very few. Let us mention in particular [ABS06], which proved that the solving

time for random 3-SAT formulas is typically polynomial up to $\alpha = 1.63$, a result in agreement yet weaker than the numerical results presented here.

22.4.3. More efficient variants of the algorithm

The threshold α_{rw} for linear time solving of random instances by PRWSAT was found above to be much smaller than the satisfiability threshold α_s . It must however be emphasized that PRWSAT is only the simplest example of a large family of local search algorithms, see for instance [SKC94, MSK97, SAO05, AA06, AAA⁺07]. They all share the same structure: a solution is searched through a random walk in the space of configurations, one variable being modified at each step. The choice of the flipped variable is made according to various heuristics; the goal is to find a compromise between the greediness of the walk which seeks to minimize locally the energy of the current assignment, and the necessity to allow for moves increasing the energy in order to avoid the trapping in local minima of the energy function. A frequently encountered ingredient of the heuristics, which is of a greedy nature, is focusing: the flipped variable necessarily belongs to at least one unsatisfied clause before the flip; this clause thus becomes satisfied after the move. Moreover, instead of choosing randomly one of the k variables of the unsatisfied clause, one can avoid flipping variables which would turn satisfied clauses into unsatisfied ones [SKC94, MSK97]. Another way to implement the greediness [SAO05] consists in bookkeeping the lowest energy found so far during the walk, and forbids flips which will raise the energy of the current assignment above the registered record plus a tolerance threshold. These demanding requirements have to be balanced with noisy, random steps, allowing to escape traps which are only locally minima of the objective function.

These more elaborate heuristics are very numerous, and depend on parameters that are finely tuned to achieve the best performances, hence an exhaustive comparison is out of the scope of this review. Let us only mention that some of these heuristics are reported in [SAO05, AA06] to efficiently find solutions of large (up to $N = 10^6$) random formulas of 3-SAT at ratio α very close to the satisfiability threshold, i.e. for $\alpha \lesssim 4.21$.

22.5. Decimation based algorithms

The algorithms studied in the remainder of the review are of a very different nature compared to the local search procedures described above. Given an initial formula F whose satisfiability has to be decided, they proceed by assigning sequentially (and permanently) the value of some of the variables. The formula can be simplified under such a partial assignment: clauses which are satisfied by at least one of their literal can be removed, while literals unsatisfying a clause are discarded from the clause. It is instructive to consider the following thought experiment: suppose one can consult an oracle who, given a formula, is able to compute the marginal probability of the variables, in the uniform probability measure over the satisfying assignments of the formula. With the help of such an oracle it would be possible to sample uniformly the satisfying assignments of F , by computing these marginals, setting one unassigned variable according to

its marginal, and then proceed in the same way with the simplified formula. A slightly less ambitious, yet still unrealistic, task is to find one satisfying configuration (not necessarily uniformly distributed) of F ; this can be performed if the oracle is able to reveal, for each formula he is questioned about, which of the unassigned variables take the same value in all satisfying assignments, and what is this value. Then it is enough to avoid setting incorrectly such a constrained variable to obtain at the end a satisfying assignment.

Of course such procedures are not meant as practical algorithms; instead of these fictitious oracles one has to resort to simplified evidences gathered from the current formula to guide the choice of the variable to assign. In Sec. 22.5.1 we consider algorithms exploiting basic information on the number of occurrences of each variable, and their behavior in the satisfiable regime of random SAT formulas. They are turned into complete algorithms by allowing for backtracking the heuristic choices, as explained in 22.5.2. Finally in Sec. 22.5.3 we shall use more refined message-passing sub-procedures to provide the information used in the assignment steps.

22.5.1. Heuristic search: the success-to-failure transition

The first algorithm we consider was introduced and analyzed by Franco and his collaborators [CF86, CF90].

1. If a formula contains a *unit clause* i.e. a clause with a single variable, this clause is satisfied through an appropriate assignment of its unique variable (propagation); If the formula contains no *unit-clause* a variable and its truth value are chosen according to some heuristic rule (free choice). Note that the unit clause propagation corresponds to the obvious answer an oracle would provide on such a formula.
2. Then the clauses in which the assigned variable appears are simplified: satisfied clauses are removed, the other ones are reduced.
3. Resume from step 1.

The procedure will end if one of two conditions is verified:

1. The formula is completely empty (all clauses have been removed), and a solution has been found (SUCCESS).
2. A contradiction is generated from the presence of two opposite unit clauses. The algorithm halts. We do not know if a solution exists and has not been found or if there is no solution (FAILURE).

The simplest example of heuristic is called Unit Clause (UC) and consists in choosing a variable uniformly at random among those that are not yet set, and assigning it to TRUE or FALSE uniformly at random. More sophisticated heuristics can take into account the number of occurrences of each variable and of its negation, the length of the clauses in which each variable appears, or they can set more than one variable at a time. For example, in the Generalized Unit Clause (GUC), the variable is always chosen among those appearing in the shortest clauses.

Numerical experiments and theory show that the results of this procedure applied to random k -SAT formulas with ratios α and size N can be classified in two regimes:

- At low ratio $\alpha < \alpha_H$ the search procedure finds a solution with positive probability (over the formulas and the random choices of the algorithm) when $N \rightarrow \infty$.
- At high ratio $\alpha > \alpha_H$ the probability of finding a solution vanishes when $N \rightarrow \infty$. Notice that $\alpha_H < \alpha_s$: solutions do exist in the range $[\alpha_H, \alpha_s]$ but are not found by this heuristic.

The above algorithm *modifies* the formula as it proceeds; during the execution of the algorithm the current formula will contain clauses of length 2 and 3 (we specialize here to $k = 3$ -SAT for the sake of simplicity but higher values of k can be considered). The sub-formulas generated by the search procedure maintain their statistical uniformity (conditioned on the number of clauses of length 2 and 3). Franco and collaborators used this fact to write down differential equations for the evolution of the densities of 2- and 3-clauses as a function of the fraction t of eliminated variables. We do not reproduce those equations here, see [Ach01] for a pedagogical review. Based on this analysis Frieze and Suen [FS96] were able to calculate, in the limit of infinite size, the probability of successful search. The outcome for the UC heuristic is

$$\mathcal{P}_{\text{success}}^{(\text{UC})}(\alpha) = \exp \left\{ -\frac{1}{4\sqrt{8/3\alpha - 1}} \arctan \left[\frac{1}{\sqrt{8/3\alpha - 1}} \right] - \frac{3}{16} \alpha \right\} \quad (22.30)$$

when $\alpha < \frac{8}{3}$, and $\mathcal{P} = 0$ for larger ratios. The probability $\mathcal{P}_{\text{success}}$ is, as expected, a decreasing function of α ; it vanishes in $\alpha_H = \frac{8}{3}$. A similar calculation shows that $\alpha_H \simeq 3.003$ for the GUC heuristic [FS96].

Franco et al's analysis can be recast in the following terms. Under the operation of the algorithm the original 3-SAT formula is turned into a mixed $2+p$ -SAT formula where p denotes the fraction of the clauses with 3 variables: there are $N\alpha \cdot (1-p)$ 2-clauses and $N\alpha p$ 3-clauses. As we mentioned earlier the simplicity of the heuristics maintains a statistical uniformity over the formulas with a given value of α and p . This observation motivated the study of the random $2+p$ -SAT ensemble by statistical mechanics methods [MZK⁺99, BMW00], some of the predictions being later proven by the rigorous analysis of [AKKK01]. At the heuristic level one expects the existence of a p dependent satisfiability threshold $\alpha_s(p)$, interpolating between the 2-SAT known threshold, $\alpha_s(p=0) = 1$, and the conjectured 3-SAT case, $\alpha_s(p=1) \approx 4.267$. The upperbound $\alpha_s(p) \leq 1/(1-p)$ is easily obtained: for the mixed formula to be satisfiable, necessarily the subformula obtained by retaining only the clauses of length 2 must be satisfiable as well. In fact this bound is tight for all values of $p \in [0, 2/5]$. During the execution of the algorithm the ratio α and the fraction p are 'dynamical' parameters, changing with the fraction $t = T/N$ of variables assigned by the algorithm. They define the coordinates of the representative point of the instance at 'time' t in the (p, α) plane of Figure 22.7. The motion of the representative point defines the search trajectory of the algorithm. Trajectories start from the point of coordinates $p(0) = 1, \alpha(0) = \alpha$ and end up on the $\alpha = 0$ axis when a solution is found. A necessary condition for the probability of success to be positive is that the 2-SAT subformula is satisfiable, that is, $\alpha \cdot (1-p) < 1$. In other words success is possible provided the trajectory does not cross the contradiction line

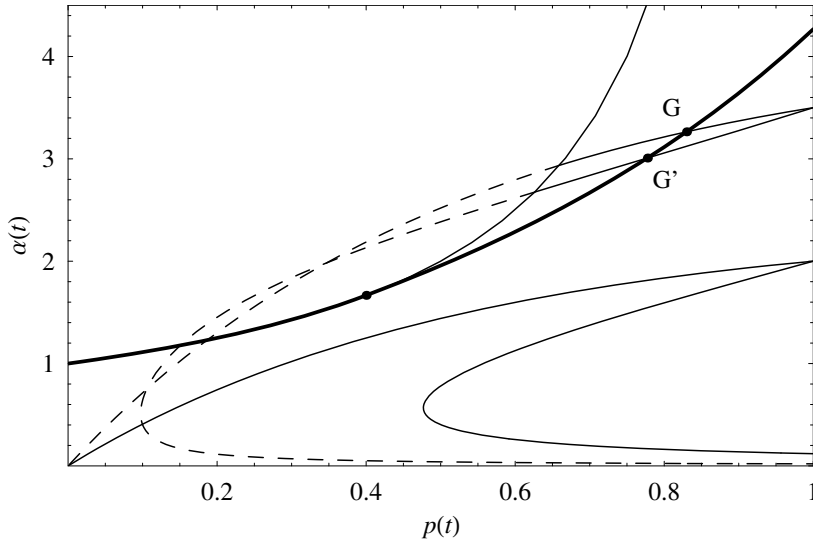


Figure 22.7. Trajectories generated by heuristic search acting on 3-SAT for $\alpha = 2$ and $\alpha = 3.5$. For all heuristics, the starting point is on the $p = 1$ axis, with the initial value of α as ordinate. The curves that end at the origin correspond to UC, those ending on the $p = 1$ axis correspond to GUC. The thick line represents the satisfiability threshold: the part on the left of the critical point $(2/5, 5/3)$ is exact and coincides with the contradiction line, where contradictions are generated with high probability, of equation $\alpha = 1/(1 - p)$, and which is plotted for larger values of p as well; the part on the right of the critical point is only a sketch. When the trajectories hit the satisfiability threshold, at points G for UC and G' for GUC, they enter a region in which massive backtracking takes place, and the trajectory represents the evolution *prior* to backtracking. The dashed part of the curves is “unphysical”, i.e. the trajectories stop when the contradiction curve is reached.

$\alpha = 1/(1 - p)$ (Figure 22.7). The largest initial ratio α such that no crossing occurs defines α_H . Notice that the search trajectory is a stochastic object. However deviations from its average locus in the plane vanish in the $N \rightarrow \infty$ limit (concentration phenomenon). Large deviations from the typical behavior can be calculated e.g. to estimate the probability of success above α_H [CM05].

The precise form of $\mathcal{P}_{\text{success}}$ and the value α_H of the ratio where it vanishes are specific to the heuristic considered (UC in (22.30)). However the behavior of the probability close to α_H is largely independent of the heuristic (provided it preserves the uniformity of the subformulas generated):

$$\ln \mathcal{P}_{\text{success}}(\alpha = \alpha_H(1 - \lambda)) \sim -\lambda^{-1/2}. \quad (22.31)$$

This universality can loosely be interpreted by observing that for α close to α_H the trajectory will pass very close to the contradiction curve $\alpha \cdot (1 - p) = 1$, which characterizes the locus of the points where the probability that a variable is assigned by the heuristics H vanishes (and all the variables are assigned by Unit Propagation). The value of α_H depend on the “shape” of the trajectory far

from this curve, and will therefore depend on the heuristics, but the probability of success (i.e. of avoiding the contradiction curve) for values of α close to α_H will only depend on the local behavior of the trajectory close to the contradiction curve, a region where most variables are assigned through Unit Propagation and not sensitive to the heuristics.

The finite-size corrections to equation (22.30) are also universal (i.e. independent on the heuristics):

$$\ln \mathcal{P}_{\text{success}}(\alpha = \alpha_H(1 - \lambda), N) \sim -N^{1/6} \mathcal{F}(\lambda N^{1/3}), \quad (22.32)$$

where \mathcal{F} is a universal scaling function which can be exactly expressed in terms of the Airy function [DM04]. This result indicates that right at α_H the probability of success decreases as a stretched exponential $\sim \exp(-cst N^{\frac{1}{6}})$.

The exponent $\frac{1}{3}$ suggests that the critical scaling of \mathcal{P} is related to random graphs. After $T = tN$ steps of the procedure, the sub-formula will consist of C_3, C_2 and C_1 clauses of length 3, 2 and 1 respectively (notice that these are *extensive*, i.e. $O(N)$ quantities). We can represent the clauses of length 1 and 2 (which are the relevant ones to understand the generation of contradictions) as an oriented graph \mathcal{G} in the following way. We will have a vertex for each literal, and represent 1-clauses by “marking” the literal appearing in each; a 2-clause will be represented by two directed edges, corresponding to the two implications equivalent to the clause (for example, $x_1 \vee \bar{x}_2$ is represented by the directed edges $\bar{x}_1 \rightarrow \bar{x}_2$ and $x_2 \rightarrow x_1$). The average out-degree of the vertices in the graph is $\gamma = C_2/(N - T) = \alpha(t)(1 - p(t))$.

What is the effect of the algorithm on \mathcal{G} ? The algorithm will proceed in “rounds”: a variable is set by the heuristics, and a series of Unit Propagations are performed until no more unit clauses are left, at which point a new round starts. Notice that during a round, extensive quantities as C_1, C_2, C_3 are likely to vary by bounded amounts and γ to vary by $O(\frac{1}{N})$ (this is the very reason that guarantees that these quantities are concentrated around their mean). At each step of Unit Propagation, a marked literal (say x) is assigned and removed from \mathcal{G} , together with all the edges connected to it, and the “descendants” of x (i.e. the literals at the end of outgoing edges) are marked. Also \bar{x} is removed together with its edges, but its descendants are not marked. Therefore, the marked vertices “diffuse” in a connected component of \mathcal{G} following directed edges. Moreover, at each step new edges corresponding to clauses of length 3 that get simplified into clauses of length 2 are added to the graph.

When $\gamma > 1$, \mathcal{G} undergoes a directed percolation transition, and a giant component of size $O(N)$ appears, in which it is possible to go from any vertex to any other vertex by following a directed path. When this happens, there is a finite probability that two opposite literals x and \bar{x} can be reached from some other literal y following a directed path. If \bar{y} is selected by Unit Propagation, at some time both x and \bar{x} will be marked, and this corresponds to a contradiction. This simple argument explains more than just the condition $\gamma = \alpha \cdot (1 - p) = 1$ for the failure of the heuristic search. It can also be used to explain the the exponent $\frac{1}{6}$ in the scaling (22.32), see [DM04, Mon07] for more details.

22.5.2. Backtrack-based search: the Davis-Putnam-Loveland-Logemann procedure

The heuristic search procedure of the previous Section can be easily turned into a complete procedure for finding solutions or proving that formulas are not satisfiable. When a contradiction is found the algorithm now backtracks to the last assigned variable (by the heuristic; unit clause propagations are merely consequences of previous assignments), invert it, and the search resumes. If another contradiction is found the algorithm backtracks to the last-but-one assigned variable and so on. The algorithm stops either if a solution is found or all possible backtracks have been unsuccessful and a proof of unsatisfiability is obtained. This algorithm was proposed by Davis, Putnam, Loveland and Logemann and is referred to as DPLL in the following.

The history of the search process can be represented by a search tree, where the nodes represent the variables assigned, and the descending edges their values (Figure 22.8). The leaves of the tree correspond to solutions (S), or to contradictions (C). The analysis of the $\alpha < \alpha_H$ regime in the previous Section leads us to the conclusion that search trees look like Figure 22.8A at small ratios⁵.

For ratios $\alpha > \alpha_H$ DPLL is very likely to find a contradiction. Backtracking enters into play, and is responsible for the drastic slowing down of the algorithm. The success-to-failure transition takes place in the non-backtracking algorithm into a polynomial-to-exponential transition in DPLL. The question is to compute the growth exponent of the average tree size, $T \sim e^{N\tau(\alpha)}$, as a function of the ratio α .

22.5.2.1. Exponential regime: Unsatisfiable formulas

Consider first the case of unsatisfiable formulas ($\alpha > \alpha_s$) where all leaves carry contradictions after DPLL halts (Figure 22.8B). DPLL builds the tree in a sequential manner, adding nodes and edges one after the other, and completing branches through backtracking steps. We can think of the same search tree built in a parallel way [CM01]. At time (depth T) our tree is composed of $L(T) \leq 2^T$ branches, each carrying a partial assignment over T variables. Step T consists in assigning one more variable to each branch, according to DPLL rules, that is, through unit-propagation or the heuristic rule. In the latter case we will speak of a splitting event, as two branches will emerge from this node, corresponding to the two possible values of the variable assigned. The possible consequences of this assignment are the emergence of a contradiction (which put an end to the branch), or the simplification of the attached formulas (the branch keeps growing).

The number of branches $L(T)$ is a stochastic variable. Its average value can be calculated as follows [Mon05]. Let us define the average number $L(\vec{C}; T)$ of branches of depth T which bear a formula containing C_3 (resp. C_2, C_1) equations of length 3 (resp. 2,1), with $\vec{C} = (C_1, C_2, C_3)$. Initially $L(\vec{C}; 0) = 1$ for $\vec{C} = (0, 0, \alpha N)$, 0 otherwise. We shall call $M(\vec{C}', \vec{C}; T)$ the average number of branches described by \vec{C}' generated from a \vec{C} branch once the T^{th} variable is

⁵A small amount of backtracking may be necessary to find the solution since $\mathcal{P}_{\text{success}} < 1$ [FS96], but the overall picture of a single branch is not qualitatively affected.

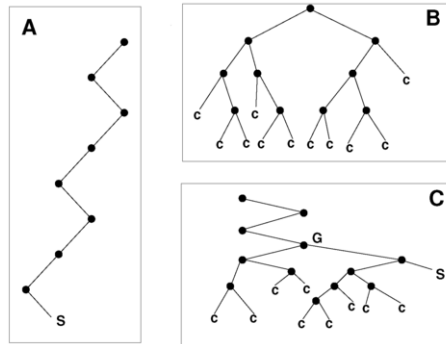


Figure 22.8. Search trees generated by DPLL: **A.** linear, satisfiable ($\alpha < \alpha_H$); **B.** exponential, unsatisfiable ($\alpha > \alpha_c$). **C.** exponential, satisfiable ($\alpha_H < \alpha < \alpha_c$); Leaves are marked with S (solutions) or C (contradictions). G is the highest node to which DPLL backtracks, see Fig. 22.7.

assigned [CM01, Mon07]. We have $0 \leq M \leq 2$, the extreme values corresponding to a contradiction and to a split respectively. We claim that

$$L(\vec{C}'; T+1) = \sum_{\vec{C}} M(\vec{C}', \vec{C}; T) L(\vec{C}; T) . \quad (22.33)$$

Evolution equation (22.33) could look like somewhat suspicious at first sight due to its similarity with the approximation we have sketched in Sec. 22.4.2 for the analysis of PRWSAT. Yet, thanks to the linearity of expectation, the correlations between the branches (or better, the instances carried by the branches) do not matter as far as the average number of branches is concerned.

For large N we expect that the number of alive (not hit by a contradiction) branches grows exponentially with the depth, or, equivalently,

$$\sum_{C_1, C_2, C_3} L(C_1, C_2, C_3; T) \sim e^{N \lambda(t) + o(N)} \quad (22.34)$$

The argument of the exponential, $\lambda(t)$, can be found using partial differential equation techniques generalizing the ordinary differential equation techniques of a single branch in the absence of backtracking (Section 22.5.1). Details can be found in [Mon05]. The outcome is that $\lambda(t)$ is a function growing from $\lambda = 0$ at $t = 0$, reaching a maximum value λ_M for some depth t_M , and decreasing at larger depths. t_M is the depth in the tree of Figure 22.8B where most contradictions are found; the number of contradiction leaves is, to exponential order, $e^{N \lambda_M}$. We conclude that the logarithm of the average size of the tree we were looking for is

$$\tau = \lambda_M . \quad (22.35)$$

For large $\alpha \gg \alpha_s$ one finds $\tau = O(1/\alpha)$, in agreement with the asymptotic scaling of [BKPS02]. The calculation can be extended to higher values of k .

22.5.2.2. Exponential regime: Satisfiable formulas

The above calculation holds for the unsatisfiable, exponential phase. How can we understand the satisfiable but exponential regime $\alpha_H < \alpha < \alpha_s$? The resolution trajectory crosses the SAT/UNSAT critical line $\alpha_s(p)$ at some point G shown in Figure 22.7. Immediately after G the instance left by DPLL is unsatisfiable. A subtree with all its leaves carrying contradictions will develop below G (Figure 22.8C). The size τ^G of this subtree can be easily calculated from the above theory from the knowledge of the coordinates (p_G, α_G) of G . Once this subtree has been built DPLL backtracks to G , flips the attached variable and will finally end up with a solution. Hence the (log of the) number of splits necessary will be equal to $\tau = (1 - t_G) \tau_{\text{split}}^G$ [CM01]. Remark that our calculation gives the logarithm of the average subtree size starting from the typical value of G . Numerical experiments show that the resulting value for τ coincides very accurately with the most likely tree size for finding a solution. The reason is that fluctuations in the sizes are mostly due to fluctuations of the highest backtracking point G , that is, of the first part of the search trajectory [CM05].

22.5.3. Message passing algorithms

According to the thought experiment proposed at the beginning of this Section valuable information could be obtained from the knowledge of the marginal probabilities of variables in the uniform measure over satisfying configurations. This is an inference problem in the graphical model associated to the formula. In this field message passing techniques (for instance Belief Propagation, or the min-sum algorithm) are widely used to compute approximately such marginals [KFL01, YFW01]. These numerical procedures introduce messages on the directed edges of the factor graph representation of the problem (recall the definitions given in Sec. 22.3.4), which are iteratively updated, the new value of a message being computed from the old values of the incoming messages (see Fig. 22.4). When the underlying graph is a tree, the message updates are guaranteed to converge in a finite number of steps, and provide exact results. In the presence of cycles the convergence of these recurrence equations is not guaranteed; they can however be used heuristically, the iterations being repeated until a fixed point has been reached (within a tolerance threshold). Though very few general results on the convergence in presence of loops are known [TJ02] (see also [MS07] for low α random SAT formulas) these heuristic procedures are often found to yield good approximation of the marginals on generic factor graph problems.

The interest in this approach for solving random SAT instances was triggered in the statistical mechanics community by the introduction of the Survey Propagation algorithm [MZ02]. Since then several generalizations and reinterpretations of SP have been put forward, see for instance [BZ04, MMW05, AGK04, Par03b, BKcvZ04, CFMZ05]. In the following paragraph we present three different message passing procedures, which differ in the nature of the messages passed between nodes, following rather closely the presentation of [BMZ05] to which we refer the reader for further details. We then discuss how these procedures have to be interleaved with assignment (decimation) steps in order to constitute a solver algorithm. Finally we shall review results obtained in a particular limit case (large

α satisfiable formulas).

22.5.3.1. Definition of the message-passing algorithms

- Belief Propagation (BP)

For the sake of readability we recall here the recursive equations (22.26) stated in Sec. 22.3.4 for the uniform probability measure over the solutions of a tree formula,

$$\begin{aligned}
 h_{i \rightarrow a} &= \sum_{b \in \partial_+ i(a)} u_{b \rightarrow i} - \sum_{b \in \partial_- i(a)} u_{b \rightarrow i} , \\
 u_{a \rightarrow i} &= -\frac{1}{2} \ln \left(1 - \prod_{j \in \partial a \setminus i} \frac{1 - \tanh h_{j \rightarrow a}}{2} \right) .
 \end{aligned} \tag{22.36}$$

where the h and u 's messages are reals (positive for u), parametrizing the marginal probabilities (beliefs) for the value of a variable in absence of some constraint nodes around it (cf. Eq. (22.25)). These equations can be used in the heuristic way explained above for any formula, and constitute the BP message-passing equations. Note that in the course of the simplification process the degree of the clauses change, we thus adopt here and in the following the natural convention that sums (resp. products) over empty sets of indices are equal to 0 (resp. 1).

- Warning Propagation (WP)

The above-stated version of the BP equations become ill-defined for an unsatisfiable formula, whether this was the case of the original formula or because of some wrong assignment steps; in particular the normalization constant of Eq. (22.24) vanishes. A way to cure this problem consists in introducing a fictitious inverse temperature β and deriving the BP equations corresponding to the regularized Gibbs-Boltzmann probability law (22.17), taking as the energy function the number of unsatisfied constraints. In the limit $\beta \rightarrow \infty$, in which the Gibbs-Boltzmann measure concentrates on the satisfying assignments, one can single out a part of the information conveyed by the BP equations to obtain the simpler Warning Propagation rules. Indeed the messages h, u are at leading order proportional to β , with proportionality coefficients we shall denote \hat{h} and \hat{u} . These messages are less informative than the ones of BP, yet simpler to handle. One finds indeed that instead of reals the WP messages are integers, more precisely $\hat{h} \in \mathbb{Z}$ and $\hat{u} \in \{0, 1\}$. They obey the following recursive equations (with a structure similar to the ones of BP),

$$\begin{aligned}
 \hat{h}_{i \rightarrow a} &= \sum_{b \in \partial_+ i(a)} \hat{u}_{b \rightarrow i} - \sum_{b \in \partial_- i(a)} \hat{u}_{b \rightarrow i} , \\
 \hat{u}_{a \rightarrow i} &= \prod_{j \in \partial a \setminus i} \mathbb{I}(\hat{h}_{j \rightarrow a} < 0) ,
 \end{aligned} \tag{22.37}$$

where $\mathbb{I}(E)$ is the indicator function of the event E . The interpretation of these equations goes as follows. $\hat{u}_{a \rightarrow i}$ is equal to 1 if in all satisfying

assignments of the amputated formula in which i is only constrained by a , i takes the value satisfying a . This happens if all other variables of clause a (i.e. $\partial a \setminus i$) are required to take their values unsatisfying a , hence the form of the right part of (22.37). In such a case we say that a sends a warning to variable i . In the first part of (22.37), the message $\hat{h}_{i \rightarrow a}$ sent by a variable to a clause is computed by pondering the number of warnings sent by all other clauses; it will in particular be negative if a majority of clauses requires i to take the value unsatisfying a .

- Survey Propagation (SP)

The convergence of BP and WP iterations is not ensured on loopy graphs. In particular the clustering phenomenon described in Sec. 22.3.1 is likely to spoil the efficiency of these procedures. The Survey Propagation (SP) algorithm introduced in [MZ02] has been designed to deal with these clustered space of configurations. The underlying idea is that the simple iterations (of BP or WP type) remain valid inside each cluster of satisfying assignments; for each of these clusters γ and each directed edge of the factor graph one has a message $h_{i \rightarrow a}^\gamma$ (and $u_{a \rightarrow i}^\gamma$). One introduces on each edge a survey of these messages, defined as their probability distribution with respect to the choice of the clusters. Then some hypotheses are made on the structure of the cluster decomposition in order to write closed equations on the survey. We now make this approach explicit in a version adapted to satisfiable instances [BMZ05], taking as the basic building block the WP equations. This leads to a rather simple form of the survey. Indeed $\hat{u}_{a \rightarrow i}$ can only take two values, its probability distribution can thus be parametrized by a single real $\delta_{a \rightarrow i} \in [0, 1]$, the probability that $\hat{u}_{a \rightarrow i} = 1$. Similarly the survey $\gamma_{i \rightarrow a}$ is the probability that $\hat{h}_{i \rightarrow a} < 0$. The second part of (22.37) is readily translated in probabilistic terms,

$$\delta_{a \rightarrow i} = \prod_{j \in \partial a \setminus i} \gamma_{j \rightarrow a} . \quad (22.38)$$

The other part of the recursion takes a slightly more complicated form,

$$\gamma_{i \rightarrow a} = \frac{(1 - \pi_{i \rightarrow a}^-) \pi_{i \rightarrow a}^+}{\pi_{i \rightarrow a}^+ + \pi_{i \rightarrow a}^- - \pi_{i \rightarrow a}^+ \pi_{i \rightarrow a}^-} ,$$

$$\text{with } \begin{cases} \pi_{i \rightarrow a}^+ = \prod_{b \in \partial_+ i(a)} (1 - \delta_{b \rightarrow i}) \\ \pi_{i \rightarrow a}^- = \prod_{b \in \partial_- i(a)} (1 - \delta_{b \rightarrow i}) \end{cases} . \quad (22.39)$$

In this equation $\pi_{i \rightarrow a}^+$ (resp. $\pi_{i \rightarrow a}^-$) corresponds to the probability that none of the clauses agreeing (resp. disagreeing) with a on the value of the literal of i sends a warning. For i to be constrained to the value unsatisfying a , at least one of the clauses of $\partial_- i(a)$ should send a warning, and none of $\partial_+ i(a)$, which explains the form of the numerator of $\gamma_{i \rightarrow a}$. The denominator arises from the exclusion of the event that both clauses in $\partial_+ i(a)$ and $\partial_- i(a)$ send messages, a contradictory event in this version of SP which is devised for satisfiable formulas.

From the statistical mechanics point of view the SP equations arise from a 1RSB cavity calculation, as sketched in Sec. 22.3.4, in the zero temperature limit ($\beta \rightarrow \infty$) and vanishing Parisi parameter m , these two limits being either taken simultaneously as in [MZ02, BKcvZ04] or successively [KMRT⁺07]. One can thus compute, from the solution of the recursive equations on a single formula, an estimation of its complexity, i.e. the number of its clusters (irrespective of their sizes). The message passing procedure can also be adapted, at the price of technical complications, to unsatisfiable clustered formulas [BKcvZ04]. Note also that the above SP equations have been shown to correspond to the BP ones in an extended configuration space where variables can take a ‘joker’ value [BZ04, MMW05], mimicking the variables which are not frozen to a single value in all the assignments of a given cluster. Interpolations between the BP and SP equations have been studied in [AGK04, MMW05].

22.5.3.2. Exploiting the information

The information provided by these message passing procedures can be exploited in order to solve satisfiability formulas; in the algorithm sketched at the beginning of Sec. 22.5.1 the heuristic choice of the assigned variable, and its truth value, can be done according to the results of the message passing on the current formula. If BP were an exact inference algorithm, one could choose any unassigned variable, compute its marginal according to Eq. (22.27), and draw it according to this probability. Of course BP is only an approximate procedure, hence a practical implementation of this idea should privilege the variables with marginal probabilities closest to a deterministic law (i.e. with the largest $|h_i|$), motivated by the intuition that these are the least subject to the approximation errors of BP. Similarly, if the message passing procedure used at each assignment step is WP, one can fix the variable with the largest $|\hat{h}_i|$ to the value corresponding to the sign of \hat{h}_i . In the case of SP, the solution of the message passing equations are used to compute, for each unassigned variable i , a triplet of numbers $(\gamma_i^+, \gamma_i^-, \gamma_i^0)$ according to

$$\gamma_i^+ = \frac{(1 - \pi_i^+) \pi_i^-}{\pi_i^+ + \pi_i^- - \pi_i^+ \pi_i^-}, \quad \gamma_i^- = \frac{(1 - \pi_i^-) \pi_i^+}{\pi_i^+ + \pi_i^- - \pi_i^+ \pi_i^-}, \quad \gamma_i^0 = 1 - \gamma_i^+ - \gamma_i^-,$$

$$\text{with } \begin{cases} \pi_i^+ = \prod_{a \in \partial_+ i} (1 - \delta_{a \rightarrow i}) \\ \pi_i^- = \prod_{a \in \partial_- i} (1 - \delta_{a \rightarrow i}) \end{cases} . \quad (22.40)$$

γ_i^+ (resp. γ_i^-) is interpreted as the fraction of clusters in which $\sigma_i = +1$ (resp. $\sigma_i = -1$) in all solutions of the cluster, hence γ_i^0 corresponds to the clusters in which σ_i can take both values. In the version of [BMZ05], one then choose the variable with the largest $|\gamma_i^+ - \gamma_i^-|$, and fix it to $\sigma_i = +1$ (resp. $\sigma_i = -1$) if $\gamma_i^+ > \gamma_i^-$ (resp. $\gamma_i^+ < \gamma_i^-$). In this way one tries to select an assignment preserving the maximal number of clusters.

Of course many variants of these heuristic rules can be devised; for instance after each message passing computation one can fix a finite fraction of the variables (instead of a single one), allows for some amount of backtracking [Par03a],

or increase a soft bias instead of assigning completely a variable [CFMZ05]. Moreover the tolerance on the level of convergence of the message passing itself can also be adjusted. All these implementation choices will affect the performances of the solver, in particular the maximal value of α up to which random SAT instances are solved efficiently, and thus makes difficult a precise statement about the limits of these algorithms. In consequence we shall only report the impressive result of [BMZ05], which presents an implementation ⁶ working for random 3-SAT instances up to $\alpha = 4.24$ (very close to the conjectured satisfiability threshold $\alpha_s \approx 4.267$) for problem sizes as large as $N = 10^7$.

The theoretical understanding of these message passing inspired solvers is still poor compared to the algorithms studied in Sec. 22.5.1, which use much simpler heuristics in their assignment steps. One difficulty is the description of the residual formula after an extensive number of variables have been assigned; because of the correlations between successive steps of the algorithm this residual formula is not uniformly distributed conditioned on a few dynamical parameters, as was the case with $(\alpha(t), p(t))$ for the simpler heuristics of Sec. 22.5.1. One version of BP guided decimation could however be studied analytically in [MRTS07], by means of an analysis of the thought experiment discussed at the beginning of Sec. 22.5. The study of another simple message passing algorithm is presented in the next paragraph.

22.5.3.3. Warning Propagation on dense random formulas

Feige proved in [Fei02] a remarkable connection between the *worst-case* complexity of approximation problems and the structure of *random* 3-SAT at large (but independent of N) values of the ratio α . He introduced the following hardness hypothesis for random 3-SAT formulas:

Hypothesis 1: *Even if α is arbitrarily large (but independent of N), there is no polynomial time algorithm that on most 3-SAT formulas outputs UNSAT, and always outputs SAT on a 3-SAT formula that is satisfiable.*

and used it to derive hardness of approximation results for various computational problems. As we have seen these instances are typically unsatisfiable; the problem of interest is thus to recognize efficiently the rare satisfiable instances of the distribution.

A variant of this problem was studied in [FMV06], where WP was proven to be effective in finding solutions of dense planted random formulas (the planted distribution is the uniform distribution conditioned on being satisfied by a given assignment). More precisely, [FMV06] proves that for α large enough (but independent of N), the following holds with probability $1 - e^{-O(\alpha)}$:

1. WP converges after at most $O(\ln N)$ iterations.
2. If a variable i has $\hat{h}_i \neq 0$, then the sign of \hat{h}_i is equal to the value of σ_i in the planted assignment. The number of such variables is bigger than $N(1 - e^{-O(\alpha)})$ (i.e. almost all variables can be reconstructed from the values of \hat{h}_i).

⁶<http://www.ictp.trieste.it/~zecchina/SP>

3. Once these variables are fixed to their correct assignments, the remaining formula can be satisfied in time $O(N)$ (in fact, it is a tree formula).

On the basis of non-rigorous statistical mechanics methods, these results were argued in [AMZ07] to remain true when the planted distribution is replaced by the uniform distribution conditioned on being satisfiable. In other words by iterating WP for a number of iterations bigger than $O(\ln N)$ one is able to detect the rare satisfiable instances at large α . The argument is based on the similarity of structure between the two distributions at large α , namely the existence of a single, small cluster of solutions where almost all variables are frozen to a given value. This correspondence between the two distributions of instances was proven rigorously in [COKV], where it was also shown that a related polynomial algorithm succeeds with high probability in finding solutions of the satisfiable distribution of large enough density α .

These results indicate that a stronger form of hypothesis 1, obtained by replacing *always* with *with probability p* (with respect to the uniform distribution over the formulas and possibly to some randomness built in the algorithm), is wrong for any $p < 1$. However, the validity of hypothesis 1 is still unknown for random 3-SAT instances. Nevertheless, this result is interesting because it is one of the rare cases in which the performances of a message-passing algorithm could be analyzed in full detail.

22.6. Conclusion

This review was mainly dedicated to the random k -Satisfiability and k -Xor-Satisfiability problems; the approach and results we presented however extend to other random decision problems, in particular random graph q -coloring. This problem consists in deciding whether each vertex of a graph can be assigned one out of q possible colors, without giving the same color to the two extremities of an edge. When input graphs are randomly drawn from Erdős-Renyi (ER) ensemble $G(N, p = c/N)$ a phase diagram similar to the one of k -SAT (Section 22.3) is obtained. There exists a colorable/uncolorable phase transition for some critical average degree $c_s(q)$, with for instance $c_s(3) \simeq 4.69$ [KPW04]. The colorable phase also exhibits the clustering and condensation transitions [ZK07] we explained on the example of the k -Satisfiability. Actually what seems to matter here is rather the structure of inputs and the symmetry properties of the decision problem rather than its specific details. All the above considered input models share a common, underlying ER random graph structure. From this point of view it would be interesting to ‘escape’ from the ER ensemble and consider more structured graphs e.g. embedded in a low dimensional space.

To what extent the similarity between phase diagrams correspond to similar behaviour in terms of hardness of resolution is an open question. Consider the case of rare satisfiable instances for the random k -SAT and k -XORSAT well above their sat/unsat thresholds (Section 22.5). Both problems share very similar statistical features. However, while a simple message-passing algorithm allows one to easily find a (the) solution for the k -SAT problem this algorithm is inefficient for random k -XORSAT. Actually the local or decimation-based algorithms

of Sections 22.4 and 22.5 are efficient to find solution to rare satisfiable instances of random k -SAT [BHL⁺02], but none of them works for random k -XORSAT (while the problem is in P!). This example raises the important question of the relationship between the statistical properties of solutions (or quasi-solutions) encoded in the phase diagram and the (average) computational hardness. Very little is known about this crucial point; on intuitive grounds one could expect the clustering phenomenon to prevent an efficient solving of formulas by local search algorithms of the random walk type. This is indeed true for a particular class of stochastic processes [MS06b], those which respect the so-called detailed balance conditions. This connection between clustering and hardness of resolution for local search algorithms is much less obvious when the detailed balance conditions are not respected, which is the case for most of the efficient variants of PRWSAT.

References

- [AA06] J. Ardelius and E. Aurell. Behavior of heuristics on large and hard satisfiability problems. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 74(3):037702, 2006.
- [AAA⁺07] M. Alava, J. Ardelius, E. Aurell, P. Kaski, S. Krishnamurthy, P. Orponen, and S. Seitz. Circumspect descent prevails in solving random constraint satisfaction problems. 2007. [arXiv:0711.4902](https://arxiv.org/abs/0711.4902).
- [ABS06] M. Alekhnovich and E. Ben-Sasson. Linear upper bounds for random walk on small density random 3-cnfs. *SIAM Journal on Computing*, 36(5):1248–1263, 2006.
- [Ach01] D. Achlioptas. Lower bounds for random 3-SAT via differential equations. *Theor. Comput. Sci.*, 265(1-2):159–185, 2001.
- [AGK04] E. Aurell, U. Gordon, and S. Kirkpatrick. Comparing beliefs, surveys, and random walks. In *NIPS*, 2004.
- [AKKK01] D. Achlioptas, L. M. Kirousis, E. Kranakis, and D. Krizanc. Rigorous results for random $(2 + p)$ -SAT. *Theor. Comput. Sci.*, 265(1-2):109–129, 2001.
- [AMRU04] A. Amraoui, A. Montanari, T. Richardson, and R. Urbanke. Finite-length scaling for iteratively decoded ldpc ensembles. [arXiv:cs.IT/0406050](https://arxiv.org/abs/cs.IT/0406050), 2004.
- [AMZ07] F. Altarelli, R. Monasson, and F. Zamponi. Can rare sat formulae be easily recognized? on the efficiency of message-passing algorithms for k-sat at large clause-to-variable ratios. *Journal of Physics A: Mathematical and Theoretical*, 40(5):867–886, 2007.
- [AP04] D. Achlioptas and Y. Peres. The threshold for random k-sat is $2k \log 2 - o(k)$. *Journal of the American Mathematical Society*, 17:947, 2004.
- [ART06] D. Achlioptas and F. Ricci-Tersenghi. On the solution-space geometry of random constraint satisfaction problems. *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, 2006. [arXiv:cs.CC/0611052](https://arxiv.org/abs/cs.CC/0611052).
- [AS00] N. Alon and J. Spencer. *The probabilistic method*. John Wiley and sons, New York, 2000.

- [BBC⁺01] B. Bollobás, C. Borgs, J. T. Chayes, J. H. Kim, and D. B. Wilson. The scaling window of the 2-SAT transition. *Random Struct. Algorithms*, 18(3):201–256, 2001.
- [BFU93] A. Z. Broder, A. M. Frieze, and E. Upfal. On the satisfiability and maximum satisfiability of random 3-cnf formulas. Number 322 in *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, 1993.
- [BHL⁺02] W. Barthel, A. K. Hartmann, M. Leone, F. Ricci-Tersenghi, M. Weigt, and R. Zecchina. Hiding solutions in random satisfiability problems: A statistical mechanics approach. *Phys. Rev. Lett.*, 88(18):188701, Apr 2002.
- [BHW03] W. Barthel, A. K. Hartmann, and M. Weigt. Solving satisfiability problems by fluctuations: The dynamics of stochastic local search algorithms. *Phys. Rev. E*, 67(6):066104, Jun 2003.
- [BKcvZ04] D. Battaglia, M. Kolář, and R. Zecchina. Minimizing energy below the glass thresholds. *Phys. Rev. E*, 70(3):036107, Sep 2004.
- [BKPS02] P. Beame, R. Karp, T. Pitassi, and M. Saks. The efficiency of resolution and davis-putnam procedures. *SIAM Journal of Computing*, 31:1048–1075, 2002.
- [BMW00] G. Biroli, R. Monasson, and M. Weigt. A variational description of the ground state structure in random satisfiability problems. *Eur. Phys. J. B*, 14:551, 2000.
- [BMZ05] A. Braunstein, M. Mézard, and R. Zecchina. Survey propagation: an algorithm for satisfiability. *Random Struct. Algorithms*, 27(2):201–226, 2005.
- [BS04] S. Baumer and R. Schuler. Improving a probabilistic 3-sat algorithm by dynamic search and independent clause pairs. *Lecture Notes in Computer Science*, 2919:150, 2004.
- [BZ04] A. Braunstein and R. Zecchina. Survey propagation as local equilibrium equations. *Journal of Statistical Mechanics: Theory and Experiment*, 2004(06):P06007, 2004.
- [CDMM03] S. Cocco, O. Dubois, J. Mandler, and R. Monasson. Rigorous decimation-based construction of ground pure states for spin-glass models on random lattices. *Phys. Rev. Lett.*, 90(4):047205, Jan 2003.
- [CF86] M.-T. Chao and J. Franco. Probabilistic analysis of two heuristics for the 3-satisfiability problem. *SIAM J. Comput.*, 15:1106–1118, 1986.
- [CF90] M. Chao and J. Franco. Probabilistic analysis of a generalization of the unit-clause literal selection heuristics for the k-satisfiability problem. *Inf. Sci.*, 51(3):289–314, 1990.
- [CFMZ05] J. Chavas, C. Furtlehner, M. Mézard, and R. Zecchina. Survey-propagation decimation through distributed local computations. *Journal of Statistical Mechanics: Theory and Experiment*, 2005(11):P11016, 2005.
- [CM01] S. Cocco and R. Monasson. Trajectories in phase diagrams, growth processes, and computational complexity: How search algorithms solve the 3-satisfiability problem. *Phys. Rev. Lett.*, 86(8):1654–1657,

- Feb 2001.
- [CM05] S. Cocco and R. Monasson. Restarts and exponential acceleration of the Davis-Putnam-Loveland-Logemann algorithm: A large deviation analysis of the generalized unit clause heuristic for random 3-SAT. *Ann. Math. Artif. Intell.*, 43(1-4):153–172, 2005.
- [COKV] A. Coja-Oghlan, M. Krivelevich, and D. Vilenchik. Why almost all k-cnf formulas are easy. to appear (2007).
- [Cov65] T. M. Cover. Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition. *IEEE Transactions on Electronic Computers*, 14:326–334, 1965.
- [Cug03] L. Cugliandolo. Dynamics of glassy systems. In J. Barrat, M. Feigelman, J. Kurchan, and J. Dalibard, editors, *Slow relaxations and nonequilibrium dynamics in condensed matter*, Les Houches, France, 2003. Springer-Verlag.
- [dIV01] W. F. de la Vega. Random 2-sat: results and problems. *Theor. Comput. Sci.*, 265(1-2):131–146, 2001.
- [DM04] C. Deroulers and R. Monasson. Critical behaviour of combinatorial search algorithms, and the unitary-propagation universality class. *EPL (Europhysics Letters)*, 68(1):153–159, 2004.
- [DM07] A. Dembo and A. Montanari. Finite size scaling for the core of large random hypergraphs. [arXiv:math.PR/0702007](https://arxiv.org/abs/math.PR/0702007), 2007.
- [DMMZ08] H. Daudé, M. Mézard, T. Mora, and R. Zecchina. Pairs of sat assignment in random boolean formulae. *Theoretical Computer Science*, 393:260–279, 2008.
- [Dub01] O. Dubois. Upper bounds on the satisfiability threshold. *Theoret. Comput. Sci.*, 265:187, 2001.
- [DZ98] A. Dembo and O. Zeitouni. *Large deviations. Theory and applications*. Springer, Berlin, 1998.
- [FA86] Y. Fu and P. W. Anderson. Application of statistical mechanics to np-complete problems in combinatorial optimisation. *Journal of Physics A: Mathematical and General*, 19(9):1605–1620, 1986.
- [Fei02] U. Feige. Relations between average case complexity and approximation complexity. In *STOC*, pages 534–543, 2002.
- [FL03] S. Franz and M. Leone. Replica bounds for optimization problems and diluted spin systems. *J. Stat. Phys.*, 111(3-4):535–564, 2003.
- [FMV06] U. Feige, E. Mossel, and D. Vilenchik. Complete convergence of message passing algorithms for some satisfiability problems. Díaz, Josep (ed.) et al., *Approximation, randomization and combinatorial optimization. Algorithms and techniques. 9th international workshop on approximation algorithms for combinatorial optimization problems, APPROX 2006, and 10th international workshop on randomization and computation, RANDOM 2006, Barcelona, Spain, August 28–30, 2006. Proceedings*. Berlin: Springer. Lecture Notes in Computer Science 4110, 339-350 (2006)., 2006.
- [Fra01] J. Franco. Results related to threshold phenomena research in satisfiability: lower bounds. *Theoret. Comput. Sci.*, 265:147, 2001.
- [Fri99] E. Friedgut. Sharp thresholds of graph properties, and the k-sat

- problem. *Journal of the American Mathematical Society*, 12:1017, 1999.
- [FS96] A. Frieze and S. Suen. Analysis of two simple heuristics on a random instance of k -SAT. *J. Algorithms*, 20(2):312–355, 1996.
- [GLBD05] P. D. Gregorio, A. Lawlor, P. Bradley, and K. A. Dawson. Exact solution of a jamming transition: Closed equations for a bootstrap percolation problem. *PNAS*, 102:5669, 2005.
- [HKP91] J. Hertz, A. Krogh, and R. Palmer. *Introduction to the theory of neural computation*. Santa Fe Institute Studies in the Science of Complexity. Addison-Wesley, Redwood city (CA), 1991.
- [Hua90] K. Huang. *Statistical Mechanics*. John Wiley and Sons, New York, 1990.
- [JLR00] S. Janson, T. Luczak, and A. Rucinski. *Random graphs*. John Wiley and Sons, New York, 2000.
- [KFL01] F. R. Kschischang, B. J. Frey, and H.-A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Trans. Inf. Theory*, 47(2):498–519, 2001.
- [KM89] W. Krauth and M. Mezard. Storage capacity of memory networks with binary couplings. *J. Physique*, 50:3057, 1989.
- [KMRT⁺07] F. Krzakala, A. Montanari, F. Ricci-Tersenghi, G. Semerjian, and L. Zdeborova. Gibbs states and the set of solutions of random constraint satisfaction problems. *Proceedings of the National Academy of Sciences*, 104(25):10318–10323, 2007.
- [KPW04] F. Krzakala, A. Pagnani, and M. Weigt. Threshold values, stability analysis, and high- q asymptotics for the coloring problem on random graphs. *Phys. Rev. E*, 70(4):046705, Oct 2004.
- [KS94] S. Kirkpatrick and B. Selman. Critical behavior in the satisfiability of random boolean expressions. *Science*, 264:1297, 1994.
- [KT87] T. R. Kirkpatrick and D. Thirumalai. p -spin-interaction spin-glass models: Connections with the structural glass problem. *Phys. Rev. B*, 36(10):5388–5397, Oct 1987.
- [Kur70] T. G. Kurtz. Solutions of ordinary differential equations as limits of pure jump Markov processes. *J. Appl. Probab.*, 7:49–58, 1970.
- [Lig85] T. M. Liggett. *Interacting particle systems*. Springer, Berlin, 1985.
- [Ma85] S. K. Ma. *Statistical Mechanics*. World Scientific, Singapore, 1985.
- [MM06] T. Mora and M. Mézard. Geometrical organization of solutions to random linear boolean equations. *Journal of Statistical Mechanics: Theory and Experiment*, 2006(10):P10007, 2006.
- [MMW05] E. Maneva, E. Mossel, and M. J. Wainwright. A new look at survey propagation and its generalizations. In *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1089–1098, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.
- [MMZ05] M. Mézard, T. Mora, and R. Zecchina. Clustering of solutions in the random satisfiability problem. *Physical Review Letters*, 94(19):197205, 2005.
- [MMZ06] S. Mertens, M. Mézard, and R. Zecchina. Threshold values of ran-

- dom K -SAT from the cavity method. *Random Struct. Algorithms*, 28(3):340–373, 2006.
- [MO94] R. Monasson and D. O’Kane. Domains of solutions and replica symmetry breaking in multilayer neural networks. *EPL (Europhysics Letters)*, 27(2):85–90, 1994.
- [Mon95] R. Monasson. Structural glass transition and the entropy of the metastable states. *Phys. Rev. Lett.*, 75(15):2847–2850, Oct 1995.
- [Mon98] R. Monasson. Optimization problems and replica symmetry breaking in finite connectivity spin glasses. *Journal of Physics A: Mathematical and General*, 31(2):513–529, 1998.
- [Mon05] R. Monasson. A generating function method for the average-case analysis of DPLL. Chekuri, Chandra (ed.) et al., Approximation, randomization and combinatorial optimization. Algorithms and techniques. 8th international workshop on approximation algorithms for combinatorial optimization problems, APPROX 2005, and 9th international workshop on randomization and computation, RANDOM 2005, Berkeley, CA, USA, August 22–24, 2005. Proceedings. Berlin: Springer. Lecture Notes in Computer Science 3624, 402-413 (2005)., 2005.
- [Mon07] R. Monasson. Introduction to phase transitions in random optimization problems. In J. Bouchaud, M. Mézard, and J. Dalibard, editors, *Complex Systems*, Les Houches, France, 2007. Elsevier.
- [MP01] M. Mézard and G. Parisi. The bethe lattice spin glass revisited. *Eur. Phys. J. B*, 20:217, 2001.
- [MP03] M. Mézard and G. Parisi. The cavity method at zero temperature. *J. Stat. Phys.*, 111(1-2):1–34, 2003.
- [MPR05] M. Mézard, M. Palassini, and O. Rivoire. Landscape of solutions in constraint satisfaction problems. *Physical Review Letters*, 95(20):200202, 2005.
- [MPRT04] A. Montanari, G. Parisi, and F. Ricci-Tersenghi. Instability of one-step replica-symmetry-broken phase in satisfiability problems. *Journal of Physics A: Mathematical and General*, 37(6):2073–2091, 2004.
- [MPV87] M. Mézard, G. Parisi, and M. A. Virasoro. *Spin glass theory and beyond*. World Scientific, Singapore, 1987.
- [MR95] R. Motwani and P. Ravaghan. *Randomized algorithms*. Cambridge University Press, Cambridge, 1995.
- [MRTS07] A. Montanari, F. Ricci-Tersenghi, and G. Semerjian. Solving constraint satisfaction problems through belief propagation-guided decimation. 2007. [arXiv:0709.1667](https://arxiv.org/abs/0709.1667), to be published in the Proceedings of the 45th Allerton Conference (2007).
- [MRTZ03] M. Mézard, F. Ricci-Tersenghi, and R. Zecchina. Two solutions to diluted p -spin models and XORSAT problems. *J. Stat. Phys.*, 111(3-4):505–533, 2003.
- [MS06a] A. Montanari and G. Semerjian. On the dynamics of the glass transition on Bethe lattices. *J. Stat. Phys.*, 124(1):103–189, 2006.
- [MS06b] A. Montanari and G. Semerjian. Rigorous inequalities between length and time scales in glassy systems. *J. Stat. Phys.*, 125(1):23–

- 54, 2006.
- [MS07] A. Montanari and D. Shah. Counting good truth assignments of random -sat formulae. In *SODA*, pages 1255–1264, 2007.
- [MSK97] D. McAllester, B. Selman, and H. Kautz. Evidence for invariants in local search. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 321–326, Providence, Rhode Island, 1997.
- [MSL92] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of sat problems. Number 459 in *Proceedings of the Tenth National Conference on Artificial Intelligence*, 1992.
- [MZ97] R. Monasson and R. Zecchina. Statistical mechanics of the random k -satisfiability model. *Phys. Rev. E*, 56(2):1357–1370, Aug 1997.
- [MZ02] M. Mézard and R. Zecchina. Random k -satisfiability problem: From an analytic solution to an efficient algorithm. *Phys. Rev. E*, 66(5):056126, Nov 2002.
- [MZ08] T. Mora and L. Zdeborova. Random subcubes as a toy model for constraint satisfaction problems. *J. Stat. Phys.*, 131:1121–1138, 2008.
- [MZK⁺99] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. 2+p-SAT: Relation of typical-case complexity to the nature of the phase transition. *Random Struct. Algorithms*, 15(3-4):414–435, 1999.
- [Pap91] C. H. Papadimitriou. On selecting a satisfying truth assignment. In *Proceedings of the 32th Annual Symposium on Foundations of Computer Science*, pages 163–169, 1991.
- [Par03a] G. Parisi. A backtracking survey propagation algorithm for k -satisfiability. 2003. [arXiv:cond-mat/0308510](https://arxiv.org/abs/cond-mat/0308510).
- [Par03b] G. Parisi. Some remarks on the survey decimation algorithm for k -satisfiability. 2003. [arXiv:cs.CC/0301015](https://arxiv.org/abs/cs.CC/0301015).
- [PS98] C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover, New York, 1998.
- [PSW96] B. Pittel, J. Spencer, and N. Wormald. Sudden emergence of a giant k -core in a random graph. *J. Comb. Theory, Ser. B*, 67(1):111–151, 1996.
- [PT04] D. Panchenko and M. Talagrand. Bounds for diluted mean-fields spin glass models. *Probab. Theory Relat. Fields*, 130(3):319–336, 2004.
- [RTWZ01] F. Ricci-Tersenghi, M. Weigt, and R. Zecchina. Simplest random k -satisfiability problem. *Phys. Rev. E*, 63(2):026702, Jan 2001.
- [SAO05] S. Seitz, M. Alava, and P. Orponen. Focused local search for random 3-satisfiability. *Journal of Statistical Mechanics: Theory and Experiment*, 2005(06):P06006, 2005.
- [Sch02] U. Schöning. A probabilistic algorithm for k -SAT based on limited local search and restart. *Algorithmica*, 32(4):615–623, January 2002.
- [Sem08] G. Semerjian. On the freezing of variables in random constraint satisfaction problems. *J. Stat. Phys.*, 130:251, 2008.
- [SKC94] B. Selman, H. A. Kautz, and B. Cohen. Noise strategies for improv-

- ing local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94)*, pages 337–343, Seattle, 1994.
- [SM03] G. Semerjian and R. Monasson. Relaxation and metastability in a local search procedure for the random satisfiability problem. *Phys. Rev. E*, 67(6):066103, Jun 2003.
- [Tal03] M. Talagrand. *Spin glasses: a challenge for mathematicians*. Springer, Berlin, 2003.
- [TJ02] S. Tatikonda and M. Jordan. Loopy belief propagation and gibbs measures. In *Proc. Uncertainty in Artificial Intell.*, volume 18, pages 493–500, 2002.
- [Wil02] D. B. Wilson. On the critical exponents of random k -SAT. *Random Struct. Algorithms*, 21(2):182–195, 2002.
- [YFW01] J. S. Yedidia, W. T. Freeman, and Y. Weiss. Bethe free energy, kikuchi approximations and belief propagation algorithms. *Advances in Neural Information Processing Systems*, 13:689, 2001.
- [YFW03] J. S. Yedidia, W. T. Freeman, and Y. Weiss. Understanding belief propagation and its generalisations. In *Exploring Artificial Intelligence in the New Millennium*, page 239, 2003.
- [ZK07] L. Zdeborová and F. Krzakala. Phase transitions in the coloring of random graphs. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 76(3):031131, 2007.

This page intentionally left blank

Chapter 23

MaxSAT, Hard and Soft Constraints

Chu Min Li and Felip Manyà

23.1. Introduction

MaxSAT is an optimization version of SAT which consists in finding an assignment that maximizes the number of satisfied clauses. It is an NP-hard problem that has shown a remarkable activity in the SAT community over the past few years, where some SAT solving techniques have been adapted to MaxSAT and incorporated into contemporary MaxSAT solvers. Examples of such SAT solving techniques include lazy data structures, variable selection heuristics, unsatisfiable core extraction, non-chronological backtracking and clause learning.

SAT solvers provide little information on unsatisfiable instances; they just report that no solution exists. However, assignments violating a minimum number of constraints, or satisfying all the hard constraints and as many soft constraints as possible, can be considered acceptable solutions in real-life scenarios. To cope with this limitation of SAT, MaxSAT, and in particular weighted MaxSAT and Partial MaxSAT, are becoming an alternative for naturally representing and efficiently solving over-constrained problems.

There are two approaches to solving MaxSAT: approximation and heuristic algorithms, which compute near-optimal solutions, and exact algorithms, which compute optimal solutions. Heuristic algorithms are fast and do not provide any guarantee about the quality of their solutions, while approximation algorithms are not so fast but provide a guarantee about the quality of their solutions.

Regarding exact algorithms, we distinguish between SAT-based algorithms and branch-and-bound algorithms. SAT-based algorithms solve MaxSAT by solving a sequence of SAT instances and are presented in a separate chapter of this handbook. The solving techniques that implement branch-and-bound algorithms are the main focus of this chapter, and are explained in detail in subsequent sections.

Nowadays, we count with remarkable results on theoretical, logical and algorithmic aspects of MaxSAT solving. Moreover, the existence of a MaxSAT evaluation, which is held annually since 2006, has been decisive for developing new MaxSAT technology and applications.

The structure of the chapter is as follows. In Section 23.2, we introduce background definitions. In Section 23.3, we present the branch and bound scheme and explain how this scheme can be improved with good quality lower bounds, clever variable selection heuristics and suitable data structures. In Section 23.4, we survey the complete logical calculus for MaxSAT defined so far, which are based on resolution and tableaux. In Section 23.5, we review the main MaxSAT approximation algorithms. In Section 23.6, we present the MaxSAT Evaluation. In Section 23.7, we review other contributions to MaxSAT that have appeared in the literature. In Section 23.8, we survey the works on MinSAT, which is the dual problem of MaxSAT. In Section 23.9, we give the conclusions and point out some future research directions.

23.2. Preliminaries

In propositional logic, a variable x_i may take values 0 (for false) or 1 (for true). A literal l_i is a variable x_i or its negation \bar{x}_i . The complementary of literal l_i is x_i if $l_i = \bar{x}_i$, and is \bar{x}_i if $l_i = x_i$. A clause is a disjunction of literals, and a CNF formula is a multiset of clauses. In MaxSAT, we represent CNF formulas as multisets of clauses instead of sets of clauses because duplicated clauses cannot be collapsed into one clause. For instance, the multiset $\{x_1, \bar{x}_1, \bar{x}_1, x_1 \vee x_2, \bar{x}_2\}$, where a clause is repeated, has a minimum of two unsatisfied clauses.

A weighted clause is a pair (C_i, w_i) , where C_i is a disjunction of literals and w_i , its weight, is a positive number, and a weighted CNF formula is a multiset of weighted clauses. The length of a (weighted) clause is the number of its literals. The size of (weighted) CNF formula ϕ , denoted by $|\phi|$, is the sum of the length of all its clauses.

An assignment of truth values to the propositional variables satisfies a literal x_i if x_i takes the value 1 and satisfies a literal \bar{x}_i if x_i takes the value 0, satisfies a clause if it satisfies at least one literal of the clause, and satisfies a CNF formula if it satisfies all the clauses of the formula. An empty clause, denoted by \square , contains no literals and cannot be satisfied. An assignment for a CNF formula ϕ is complete if all the variables occurring in ϕ have been assigned; otherwise, it is partial.

The MaxSAT problem for a CNF formula ϕ is the problem of finding an assignment that maximizes the number of satisfied clauses. In this sequel we often use the term MaxSAT meaning MinUNSAT. This is because, with respect to exact computations, finding an assignment that minimizes the number of unsatisfied clauses is equivalent to finding an assignment that maximizes the number of satisfied clauses. Notice that an upper (lower) bound in MinUNSAT is greater (smaller) than or equal to the minimum number of clauses that can be unsatisfied by an interpretation. MaxSAT is called Max- k SAT when all the clauses have at most k literals per clause.

MaxSAT instances ϕ_1 and ϕ_2 are equivalent if ϕ_1 and ϕ_2 have the same number of unsatisfied clauses for every complete assignment of ϕ_1 and ϕ_2 .

We will also consider three extensions of MaxSAT which are more well-suited for representing and solving over-constrained problems: Weighted MaxSAT, Partial MaxSAT, and weighted Partial MaxSAT.

The weighted MaxSAT problem for a weighted CNF formula ϕ is the problem of finding an assignment that maximizes the sum of weights of satisfied clauses (or equivalently, that minimizes the sum of weights of unsatisfied clauses).

The Partial MaxSAT problem for a CNF formula, in which some clauses are declared to be *relaxable* or *soft* and the rest are declared to be *non-relaxable* or *hard*, is the problem of finding an assignment that satisfies all the hard clauses and the maximum number of soft clauses.

The weighted Partial MaxSAT problem is the combination of Partial MaxSAT and weighted MaxSAT. In weighted Partial MaxSAT, soft clauses are weighted, and solving a weighted Partial MaxSAT instance amounts to find an assignment that satisfies all the hard clauses and maximizes the sum of weights of satisfied soft clauses (or equivalently, that minimizes the sum of weights of unsatisfied soft clauses).

Notice that MaxSAT could be defined as weighted MaxSAT restricted to formulas whose clauses have weight 1, and as Partial MaxSAT in the case that all the clauses are declared to be soft.

Finally, we introduce the integer linear programming (ILP) formulation of weighted MaxSAT, which is used to compute lower bounds and upper bounds. Let $\phi = \{(C_1, w_1), \dots, (C_m, w_m)\}$ be a weighted MaxSAT instance over the propositional variables x_1, \dots, x_n . With each propositional variable x_i , we associate a variable $y_i \in \{0, 1\}$ such that $y_i = 1$ if variable x_i is true and $y_i = 0$, otherwise. With each clause C_j , we associate a variable $z_j \in \{0, 1\}$ such that $z_j = 1$ if clause C_j is satisfied and $z_j = 0$, otherwise. Let I_j^+ be the set of indices of unnegated variables in clause C_j , and let I_j^- be the set of indices of negated variables in clause C_j . The ILP formulation of the weighted MaxSAT instance ϕ is defined as follows:

$$\max F(y, z) = \sum_{j=1}^m w_j z_j$$

subject to

$$\sum_{i \in I_j^+} y_i + \sum_{i \in I_j^-} (1 - y_i) \geq z_j \quad j = 1, \dots, m$$

$$y_i \in \{0, 1\} \quad i = 1, \dots, n$$

$$z_j \in \{0, 1\} \quad j = 1, \dots, m$$

Assume now that, with each clause C_j , we associate a variable $z_j \in \{0, 1\}$ such that $z_j = 1$ if clause C_j is unsatisfied and $z_j = 0$, otherwise. Then, the ILP formulation of the minimization version of weighted MaxSAT (i.e.; weighted MinUNSAT) for the instance ϕ is defined as follows:

$$\min F(y, z) = \sum_{j=1}^m w_j z_j$$

subject to

$$\sum_{i \in I_j^+} y_i + \sum_{i \in I_j^-} (1 - y_i) + z_j \geq 1 \quad j = 1, \dots, m$$

$$y_i \in \{0, 1\} \quad i = 1, \dots, n$$

$$z_j \in \{0, 1\} \quad j = 1, \dots, m$$

The linear programming (LP) relaxation of both formulations is obtained by allowing the integer variables to take real values in $[0, 1]$.

Ansótegui and Gabàs [AG13] reported an extensive empirical investigation that indicates that solving MaxSAT instances by translating them into ILP and applying a Mixed Integer Programming (MIP) solver is competitive on crafted instances.

Weighted Partial MinSAT for an instance ϕ is the problem of finding an assignment in which the sum of the weights of the satisfied soft clauses is minimal, and all the hard clauses are satisfied. Weighted MinSAT is Weighted Partial MinSAT when there are no hard clauses. Partial MinSAT is Weighted Partial MinSAT when all the soft clauses have the same weight. MinSAT is Partial MinSAT when there are no hard clauses.

23.3. Branch and Bound Algorithms

There are competitive exact MaxSAT solvers—as the ones developed by [AH14a, AMP03, AMP05, AMP08, HLO08, Kue10, LHdG08, LMP07, LS07, LSL08, PPC⁺08, RG07, SZ04, XZ04, XZ05, ZSM03]—that implement variants of the following branch and bound (BnB) scheme for solving the minimization version of MaxSAT: Given a MaxSAT instance ϕ , BnB explores the search tree that represents the space of all possible assignments for ϕ in a depth-first manner. At every node, BnB compares the upper bound (UB), which is the best solution found so far for a complete assignment, with the lower bound (LB), which is the sum of the number of clauses which are unsatisfied by the current partial assignment plus an underestimation of the number of clauses that will become unsatisfied if the current partial assignment is completed. If $LB \geq UB$, the algorithm prunes the subtree below the current node and backtracks chronologically to a higher level in the search tree. If $LB < UB$, the algorithm tries to find a better solution by extending the current partial assignment by instantiating one more variable. The optimal number of unsatisfied clauses in the input MaxSAT instance is the value that UB takes after exploring the entire search tree.

Figure 23.1 shows the pseudo-code of a basic solver for MaxSAT. We use the following notation:

- $simplifyFormula(\phi)$ is a procedure that transforms ϕ into an equivalent and simpler instance by applying inference rules.
- $\#emptyClauses(\phi)$ is a function that returns the number of empty clauses in ϕ .
- LB is a lower bound of the minimum number of unsatisfied clauses in ϕ if the current partial assignment is extended to a complete assignment. We assume that its initial value is 0.
- $underestimation(\phi)$ is a function that returns an underestimation of the minimum number of non-empty clauses in ϕ that will become unsatisfied if the current partial assignment is extended to a complete assignment.

Require: $MaxSAT(\phi, UB)$: A MaxSAT instance ϕ and an upper bound UB

- 1: $\phi \leftarrow simplifyFormula(\phi)$;
- 2: **if** $\phi = \emptyset$ or ϕ only contains empty clauses **then**
- 3: return $\#emptyClauses(\phi)$;
- 4: **end if**
- 5: $LB \leftarrow \#emptyClauses(\phi) + underestimation(\phi)$;
- 6: **if** $LB \geq UB$ **then**
- 7: return UB ;
- 8: **end if**
- 9: $x \leftarrow selectVariable(\phi)$;
- 10: $UB \leftarrow \min(UB, MaxSAT(\phi_x, UB))$;
- 11: return $\min(UB, MaxSAT(\phi_{\bar{x}}, UB))$;

Ensure: The minimal number of unsatisfied clauses in ϕ

Figure 23.1. A basic branch and bound algorithm for MaxSAT

- UB is an upper bound of the number of unsatisfied clauses in an optimal solution. An elementary initial value for UB is the total number of clauses in the input formula, or the number of clauses which are unsatisfied by an arbitrary interpretation. Another alternative is to solve the LP relaxation of the ILP formulation of the input instance and take as upper bound the number of unsatisfied clauses in the interpretation obtained by rounding variable y_i , for $1 \leq i \leq n$, to an integer solution in a randomized way by interpreting the values of $y_i \in [0, 1]$ as probabilities (set propositional variable x_i to true with probability y_i , and set propositional variable x_i to false with probability $1 - y_i$). Nevertheless, most of the solvers take as initial upper bound the number of unsatisfied clauses that can be detected by executing the input formula in a local search solver during a short period of time.
- $selectVariable(\phi)$ is a function that returns a variable of ϕ following an heuristic.
- ϕ_x ($\phi_{\bar{x}}$) is the formula obtained by setting the variable x to true (false); i.e., by applying the one-literal rule to ϕ using the literal x (\bar{x}).

Modern MaxSAT solvers implement the basic algorithm augmented with powerful inference techniques, good quality lower bounds, clever variable selection heuristics, and efficient data structures. Partial MaxSAT solvers are also augmented with learning of hard clauses, and non-chronological backtracking.

23.3.1. Improving the Lower Bound with Underestimations

The simplest method to compute a lower bound, when solving the minimization version of MaxSAT, consists in just counting the number of clauses which are unsatisfied by the current partial assignment [BF99]. One step forward is to incorporate an underestimation of the number of clauses that will become unsatisfied if the current partial assignment is extended to a complete assignment. The most basic method was defined by Wallace and Freuder [WF96]:

$$LB(\phi) = \#emptyClauses(\phi) + \sum_{x \text{ occurs in } \phi} \min(ic(x), ic(\bar{x})),$$

where ϕ is the CNF formula associated with the current partial assignment, and $ic(x)$ ($ic(\bar{x})$) —inconsistency count of x (\bar{x})— is the number of unit clauses of ϕ that contain \bar{x} (x). In other words, that underestimation is the number of disjoint inconsistent subformulas in ϕ formed by a unit clause with a literal l and a unit clause with the complementary of l .

Lower bounds dealing with longer clauses include the star rule and UP. In the star rule [SZ04, AMP04], the underestimation of the lower bound is the number of disjoint inconsistent subformulas of the form $\{l_1, \dots, l_k, \bar{l}_1 \vee \dots \vee \bar{l}_k\}$. When $k = 1$, the star rule is equivalent to the inconsistency counts of Wallace and Freuder.

In UP [LMP05], the underestimation of the lower bound is the number of disjoint inconsistent subformulas that can be detected with unit propagation. UP works as follows: It applies unit propagation until a contradiction is derived. Then, UP identifies, by inspecting the implication graph, a subset of clauses from which a unit refutation can be constructed, and tries to identify new contradictions from the remaining clauses. The order in which unit clauses are propagated has a clear impact on the quality of the lower bound [LMP06].

UP can be enhanced with failed literal detection as follows: Given a MaxSAT instance ϕ and a variable x occurring positively and negatively in ϕ , UP is applied to both $\phi \wedge \{x\}$ and $\phi \wedge \{\bar{x}\}$. If UP derives a contradiction from $\phi \wedge \{x\}$ and another contradiction from $\phi \wedge \{\bar{x}\}$, then the union of the two inconsistent subsets identified by UP, once we have removed the unit clauses x and \bar{x} , is an inconsistent subset of ϕ . UP enhanced with failed literal detection does not need the occurrence of unit clauses in the input formula for deriving a contradiction. While UP only identifies unit refutations, UP enhanced with failed literal detection identifies non-unit refutations too. Since applying failed literal detection to every variable is time consuming, it is applied to a reduced number of variables in practice [LMP06].

MaxSAT solvers like ahmaxsat [AH14a], MaxSatz [LMP07], and Mini-MaxSat [HLO08] apply either UP or UP enhanced with failed literal detection. Nowadays, UP-based lower bounds are the prevailing approach to computing underestimations in branch-and-bound MaxSAT solvers. This technique has also been applied to solve the maximum clique problem [JLM17, LFX13, LJM17, LJX15, LQ10].

Darras et al. [DDDL07] developed a version of UP in which the computation of the lower bound is made more incremental by saving some of the small size disjoint inconsistent subformulas detected by UP. They avoid to redetect the saved inconsistencies if they remain in subsequent nodes of the proof tree, and are able to solve some types of instances faster. Lin et al. [LSL08] defined an improved version of UP that, besides being incremental, guarantees that the lower bound computed at a node of the search tree is not smaller than the lower bound computed at the parent of that node. Abramé and Habet [AH14c] proposed an improved implementation of UP in ahmaxsat that undoes propagations in a non-chronological order and can produce smaller inconsistent subsets. Shen and Zhang [SZ04] defined a lower bound, called LB4, which is similar to UP but restricted to Max-2SAT instances and using a static variable ordering.

A variant of UP enhanced with failed literal detection was implemented in the solver akmaxsat [Kue10]. It can be the case that UP derives a contradiction

from $\phi \wedge \{l\}$ but not from $\phi \wedge \{\bar{l}\}$. In fact, this shows that \bar{l} follows from ϕ . If ϕ' is the result of applying UP to $\phi \wedge \{\bar{l}\}$, then the algorithm tries to find another failed literal l' in ϕ' . If UP derives a contradiction from both $\phi \wedge \{l'\}$ and $\phi \wedge \{\bar{l}'\}$, the algorithm stops and identifies an inconsistent subset. If UP derives a contradiction from $\phi \wedge \{l'\}$ but not from $\phi \wedge \{\bar{l}'\}$, the same process is repeated on the formula resulting of applying UP to $\phi \wedge \{\bar{l}'\}$ until an inconsistent subset is detected or no more failed literals can be found.

Another approach to computing underestimation is based on first reducing the MaxSAT instance to be solved to an instance of another problem, and then solve a relaxation of the obtained instance. For example, the solvers Clone [PD07, PPC⁺08] and SR(w) [RG07], reduce MaxSAT to the minimum cardinality problem. Since the minimum cardinality problem is NP-hard for a CNF formula ϕ and can be solved in time linear in the size of a deterministic decomposable negation normal form (d-DNNF) compilation of ϕ , Clone and SR(w) solve the minimum cardinality problem of a d-DNNF compilation of a relaxation of ϕ . The worst-case complexity of a d-DNNF compilation of ϕ is exponential in the treewidth of its constraint graph, and Clone and SR(w) obtain a relaxation of ϕ with bounded treewidth by renaming different occurrences of some variables.

Xing and Zhang [XZ05] reduce the MaxSAT instance to the ILP formulation of the minimization version of MaxSAT (c.f. Section 23.2), and then solve the LP relaxation. An optimal solution of the LP relaxation provides an underestimation of the lower bound because the LP relaxation is less restricted than the ILP formulation. In practice, they apply that lower bound computation method only to nodes containing unit clauses. If each clause in the MaxSAT instance has more than one literal, then $y_i = \frac{1}{2}$ for all $1 \leq i \leq n$ and $z_j = 0$ for all $1 \leq j \leq m$ is an optimal solution of the LP relaxation. In this case, the underestimation is 0. Nevertheless, LP relaxations do not seem to be as competitive as the rest of approaches.

23.3.2. Improving the Lower Bound with Inference

Another approach to improve the quality of the lower bound consists in applying inference rules that transform a MaxSAT instance ϕ into an *equivalent* but simpler MaxSAT instance ϕ' . In the best case, inference rules produce new empty clauses in ϕ' that allow to increment the lower bound. In contrast with the empty clauses derived when computing underestimations, the empty clauses derived with inference rules do not have to be recomputed at every node of the current subtree so that the lower bound computation is more incremental.

A MaxSAT inference rule is *sound* if it transforms an instance ϕ into an equivalent instance ϕ' . It is not sufficient to preserve satisfiability as in SAT, ϕ and ϕ' must have the same number of unsatisfied clauses for every possible assignment. Unfortunately, unit propagation, which is the most powerful inference technique applied in DPLL-style SAT solvers, is unsound for MaxSAT as the next example shows: The set of clauses $\{x_1, \bar{x}_1 \vee x_2, \bar{x}_1 \vee \bar{x}_2, \bar{x}_1 \vee x_3, \bar{x}_1 \vee \bar{x}_3\}$ has a minimum of one unsatisfied clause (setting x_1 to false), but two empty clauses are derived by applying unit propagation.

MaxSAT inference rules are also called *transformation rules* in the literature

because the premises of the rule are replaced with the conclusion when a rule is applied. If the conclusion is added to the premises as in SAT, the number of clauses which are unsatisfied by an assignment might increase.

The amount of inference enforced by existing BnB MaxSAT solvers at each node of the proof tree is poor compared with the inference enforced by DPLL-style SAT solvers. The simplest inference enforced, when branching on a literal l , consists in applying the one-literal rule: The clauses containing l are removed from the instance and the occurrences of \bar{l} are removed from the clauses in which \bar{l} appears, but the existing unit clauses and the new unit clauses derived as a consequence of removing the occurrences of \bar{l} are not propagated as in unit propagation. That inference is typically enhanced with the MaxSAT inference rules described in the rest of this section.

First, we present simple inference rules that have proved to be useful in a number of solvers [AMP03, AMP05, BF99, SZ04, XZ05], and then some more sophisticated inferences rules which are implemented in solvers like ahmaxsat [AH14a], akmaxsat [Kue10], MaxSatz [LMP07], and MiniMaxSat [HLO08]. Some simple inference rules are:

- The pure literal rule [BF99]: If a literal only appears with either positive or negative polarity in a MaxSAT instance, all the clauses containing that literal are removed.
- The dominating unit clause rule [NR00]: If the number of clauses (of any length) in which a literal l appears is not greater than the number of unit clauses in which \bar{l} appears, all the clauses containing l and all the occurrences of \bar{l} are removed.
- The complementary unit clause rule [NR00]: If a MaxSAT instance contains a unit clause with the literal l and a unit clause with the literal \bar{l} , these two clauses are replaced with one empty clause.
- The almost common clause rule [BR99]: If a MaxSAT instance contains a clause $x \vee D$ and a clause $\bar{x} \vee D$, where D is a disjunction of literals, then both clauses are replaced with D . In practice, this rule is applied when D contains at most one literal.

The resolution rule applied in SAT (i.e., derive $D \vee D'$ from $x \vee D$ and $\bar{x} \vee D'$) preserves satisfiability but not equivalence, and therefore cannot be applied to MaxSAT instances, except for some particular cases like the almost common clause rule. We refer the reader to Section 23.4 for a complete resolution calculus for MaxSAT, and devote the rest of this section to present some sound MaxSAT resolution rules that can be applied in polynomial time.

We start by presenting the *star rule*: If $\phi_1 = \{l_1, \bar{l}_1 \vee \bar{l}_2, l_2\} \cup \phi'$, then $\phi_2 = \{\square, l_1 \vee l_2\} \cup \phi'$ is equivalent to ϕ_1 . This rule, which can be seen as the inference counterpart of the underestimation of the same name, can also be presented as follows:

$$\left\{ \begin{array}{l} l_1 \\ \bar{l}_1 \vee \bar{l}_2 \\ l_2 \end{array} \right\} \Longrightarrow \left\{ \begin{array}{l} \square \\ l_1 \vee l_2 \end{array} \right\} \quad (23.1)$$

Notice that the rule detects a contradiction from $l_1, \bar{l}_1 \vee \bar{l}_2, l_2$ and, therefore,

replaces these clauses with an empty clause. In addition, the rule adds the clause $l_1 \vee l_2$ to ensure the equivalence between ϕ_1 and ϕ_2 . For any assignment containing either $l_1 = 0, l_2 = 1$, or $l_1 = 1, l_2 = 0$, or $l_1 = 1, l_2 = 1$, the number of unsatisfied clauses in $\{l_1, \bar{l}_1 \vee \bar{l}_2, l_2\}$ is 1, but for any assignment containing $l_1 = 0, l_2 = 0$, the number of unsatisfied clauses is 2. Notice that even when any assignment containing $l_1 = 0, l_2 = 0$ is not the best assignment for the subset $\{l_1, \bar{l}_1 \vee \bar{l}_2, l_2\}$, it can be the best for the whole formula. By adding $l_1 \vee l_2$, the rule ensures that the number of unsatisfied clauses in ϕ_1 and ϕ_2 is also the same when $l_1 = 0, l_2 = 0$.

This rule can be generalized in such a way that it captures unit resolution refutations in which clauses and resolvents are used exactly once:

$$\left\{ \begin{array}{c} l_1 \\ \bar{l}_1 \vee l_2 \\ \bar{l}_2 \vee l_3 \\ \dots \\ \bar{l}_k \vee l_{k+1} \\ \bar{l}_{k+1} \end{array} \right\} \Longrightarrow \left\{ \begin{array}{c} \square \\ l_1 \vee \bar{l}_2 \\ l_2 \vee \bar{l}_3 \\ \dots \\ l_k \vee \bar{l}_{k+1} \end{array} \right\} \quad (23.2)$$

The last two rules consume two unit clauses for deriving one contradiction. Next, we define two inference rules that capture unit resolution refutations in which (i) exactly one unit clause is consumed, and (ii) the unit clause is used twice in the derivation of the empty clause. The second rule is a combination of the first rule with a linear derivation.

$$\left\{ \begin{array}{c} l_1 \\ \bar{l}_1 \vee l_2 \\ \bar{l}_1 \vee l_3 \\ \bar{l}_2 \vee \bar{l}_3 \end{array} \right\} \Longrightarrow \left\{ \begin{array}{c} \square \\ l_1 \vee \bar{l}_2 \vee \bar{l}_3 \\ \bar{l}_1 \vee l_2 \vee l_3 \end{array} \right\} \quad (23.3)$$

$$\left\{ \begin{array}{c} l_1 \\ \bar{l}_1 \vee l_2 \\ \bar{l}_2 \vee l_3 \\ \dots \\ \bar{l}_k \vee l_{k+1} \\ \bar{l}_{k+1} \vee l_{k+2} \\ \bar{l}_{k+1} \vee l_{k+3} \\ \bar{l}_{k+2} \vee \bar{l}_{k+3} \end{array} \right\} \Longrightarrow \left\{ \begin{array}{c} \square \\ l_1 \vee \bar{l}_2 \\ l_2 \vee \bar{l}_3 \\ \dots \\ l_k \vee \bar{l}_{k+1} \\ l_{k+1} \vee \bar{l}_{k+2} \vee \bar{l}_{k+3} \\ \bar{l}_{k+1} \vee l_{k+2} \vee l_{k+3} \end{array} \right\} \quad (23.4)$$

MaxSatz implements the almost common clause rule, Rule 23.1, Rule 23.2, Rule 23.3 and Rule 23.4. Some of these rules are also applied in the solvers *ahmaxsat* and *akmaxsat*.

Independently and in parallel to the definition of the rules of MaxSatz, similar inference rules were defined for weighted MaxSAT by Heras and Larrosa [LH05, HL06], and were implemented in Max-DPLL [LHdG08]. These rules were inspired by the soft local consistency properties defined in the constraint programming community [dGLMS03]. The rules implemented in Max-DPLL are the almost common clause rule, chain resolution and cycle resolution. Chain resolution,

which allows one to derive a new empty clause, is defined as follows:

$$\left\{ \begin{array}{l} (l_1, w_1), \\ (\bar{l}_i \vee l_{i+1}, w_{i+1})_{1 \leq i < k}, \\ (\bar{l}_k, w_{k+1}) \end{array} \right\} \Longrightarrow \left\{ \begin{array}{l} (l_i, m_i - m_{i+1})_{1 \leq i \leq k}, \\ (\bar{l}_i \vee l_{i+1}, w_{i+1} - m_{i+1})_{1 \leq i < k}, \\ (l_i \vee \bar{l}_{i+1}, m_{i+1})_{1 \leq i < k}, \\ (\bar{l}_k, w_{k+1} - m_{k+1}), \\ (\square, m_{k+1}) \end{array} \right\} \quad (23.5)$$

where w_i , $1 \leq i \leq k+1$, is the weight of the corresponding clause, and $m_i = \min(w_1, w_2, \dots, w_i)$. Chain resolution is equivalent to Rule 23.2 if it is applied to unweighted MaxSAT.

Cycle resolution, which allows one to derive a new unit clause and whose application is restricted to $k = 3$ in Max-DPLL, is defined as follows:

$$\left\{ \begin{array}{l} (\bar{l}_i \vee l_{i+1}, w_i)_{1 \leq i < k}, \\ (\bar{l}_1 \vee \bar{l}_k, w_k) \end{array} \right\} \Longrightarrow \left\{ \begin{array}{l} (\bar{l}_1 \vee l_i, m_{i-1} - m_i)_{2 \leq i \leq k}, \\ (\bar{l}_i \vee l_{i+1}, w_i - m_i)_{2 \leq i < k}, \\ (\bar{l}_1 \vee l_i \vee \bar{l}_{i+1}, m_i)_{2 \leq i < k}, \\ (l_1 \vee \bar{l}_i \vee l_{i+1}, m_i)_{2 \leq i < k}, \\ (\bar{l}_1 \vee \bar{l}_k, w_k - m_k), \\ (\bar{l}_1, m_k) \end{array} \right\} \quad (23.6)$$

An analysis of the impact of cycle resolution on the performance of MaxSAT solvers can be found in [LMMP08, LMMP09, LMMP10].

A more general inference scheme is implemented in MiniMaxSat [HLO07, HLO08]. It detects a contradiction with unit propagation and identifies an unsatisfiable subset. Then, it creates a refutation for that unsatisfiable subset and applies the MaxSAT resolution rule defined in Section 23.4 if the size of the largest resolvent in the refutation is smaller than 4.

The lower bound computation methods based on unit propagation represent the different derivations of unit clauses in a graph, called implication graph [LMP07]. Looking at that graph, solvers identify the clauses which are involved in the derivation of a contradiction. In contemporary MaxSAT solvers, this graph is also used to decide whether the clauses involved in a contradiction match with the premises of the above mentioned inference rule.

Abramé and Habet [AH15b] showed that in some cases it is better not to apply MaxSAT resolution to certain inconsistent subsets of clauses because the transformations derived do not allow to detect further inconsistent subsets. They also showed that in other cases further inconsistent subsets can be detected if MaxSAT resolution is applied locally [AH14b].

23.3.3. Variable Selection Heuristics

Most of the exact MaxSAT solvers incorporate variable selection heuristics that take into account the number of literal occurrences in such a way that each occurrence has an associated weight that depends on the length of the clause that contains the literal. MaxSAT heuristics give priority to literals occurring in binary clauses instead of literals occurring in unit clauses as SAT heuristics do.

Let us see as an example the variable selection heuristic of MaxSatz [LMP07]: Let $neg1(x)$ ($pos1(x)$) be the number of unit clauses in which x is negative (positive), $neg2(x)$ ($pos2(x)$) be the number of binary clauses in which x is negative (positive), and let $neg3(x)$ ($pos3(x)$) be the number of clauses containing three or more literals in which x is negative (positive). MaxSatz selects the variable x such that $(neg1(x) + 4 * neg2(x) + neg3(x)) * (pos1(x) + 4 * pos2(x) + pos3(x))$ is the largest. Once a variable x is selected, MaxSatz applies the following value selection heuristic: If $neg1(x) + 4 * neg2(x) + neg3(x) < pos1(x) + 4 * pos2(x) + pos3(x)$, set x to true; otherwise, set x to false. The solver `ahmaxsat` [AH14a] implements a variant of this variable selection heuristic.

Earlier MaxSAT solvers (AMP [AMP03], Lazy [AMP05], MaxSolver [XZ05], Max-DPLL [LHdG08], ...) incorporate variants of the two-sided Jeroslow rule that give priority to variables occurring often in binary clauses. MaxSolver changes the weights as the search proceeds.

23.3.4. Data Structures

Data structures for SAT have been naturally adapted to MaxSAT. We can divide the solvers into two classes: solvers like `ahmaxsat`, `akmaxsat` and MaxSatz representing formulas with adjacency lists, and solvers like Lazy and MiniMaxSat which use data structures with watched literals. Lazy data structures are particularly good when there is a big number of clauses; for example, in Partial MaxSAT solvers with clause learning.

23.4. Complete Inference in MaxSAT

23.4.1. MaxSAT Resolution

A natural extension to MaxSAT of the resolution rule applied in SAT was defined by Larrosa and Heras [LH05]:

$$\frac{\begin{array}{c} x \vee A \\ \bar{x} \vee B \end{array}}{\frac{A \vee B}{x \vee A \vee \bar{B}}} \\ \bar{x} \vee \bar{A} \vee B$$

However, two of the conclusions of this rule are not in clausal form, and the application of distributivity results into an unsound rule: Assume that $A = a_1$, and $B = b_1 \vee b_2$. Then, the interpretation that assigns false to x and a_1 , and true to b_1 and b_2 satisfies one clause from the premises ($x \vee a_1$) and unsatisfies two clauses from the conclusion of the rule ($x \vee a_1 \vee \bar{b}_1, x \vee a_1 \vee \bar{b}_2$). Since the rule does not preserve the number of unsatisfied clauses, it is unsound.

Independently and in parallel, Bonet et al. [BLM06, BLM07], and Heras and Larrosa [HL06] defined a sound version of the previous rule with the conclusions in clausal form:

$$\begin{array}{c}
x \vee a_1 \vee \cdots \vee a_s \\
\overline{x} \vee b_1 \vee \cdots \vee b_t \\
\hline
a_1 \vee \cdots \vee a_s \vee b_1 \vee \cdots \vee b_t \\
x \vee a_1 \vee \cdots \vee a_s \vee \overline{b_1} \\
x \vee a_1 \vee \cdots \vee a_s \vee b_1 \vee \overline{b_2} \\
\cdots \\
x \vee a_1 \vee \cdots \vee a_s \vee b_1 \vee \cdots \vee b_{t-1} \vee \overline{b_t} \\
\overline{x} \vee b_1 \vee \cdots \vee b_t \vee \overline{a_1} \\
\overline{x} \vee b_1 \vee \cdots \vee b_t \vee a_1 \vee \overline{a_2} \\
\cdots \\
\overline{x} \vee b_1 \vee \cdots \vee b_t \vee a_1 \vee \cdots \vee a_{s-1} \vee \overline{a_s}
\end{array}$$

This inference rule concludes, apart from the conclusion where a variable has been cut, some additional clauses that contain one of the premises as subclause. We say that the rule *cuts* the variable x . The tautologies concluded by the rule are removed, and the repeated literals in a clause are collapsed into one.

Notice that an instance of MaxSAT resolution not only depends on the two premises and the cut variable (like in resolution), but also on the order of the literals in the premises. Notice also that, like in resolution, this rule concludes a new clause not containing the variable x , except when this clause is a tautology.

Bonet et al. [BLM06, BLM07] proved the completeness of MaxSAT resolution: By *saturating* successively w.r.t. all the variables, one derives as many empty clauses as the minimum number of unsatisfied clauses in the MaxSAT input instance. Saturating w.r.t. a variable amounts to apply the MaxSAT resolution rule to clauses containing that variable until every possible application of the inference rule only introduces clauses containing that variable (since tautologies are eliminated). Once a MaxSAT instance is saturated w.r.t. a variable, all the clauses containing that variable are not considered to saturate w.r.t. another variable. We refer to [BLM07] for further technical details and for the weighted version of the rule.

We consider the multiset of clauses $\phi = \{\overline{x}_1, x_1 \vee x_2, x_1 \vee x_3, \overline{x}_3\}$ to illustrate how a variable saturation algorithm works. We start by considering variable x_1 and resolve the first two clauses, obtaining $\{x_2, \overline{x}_1 \vee \overline{x}_2, x_1 \vee x_3, \overline{x}_3\}$. We then resolve the second and third clause and get a saturation of ϕ w.r.t. x_1 : $\{x_2, \overline{x}_2 \vee x_3, \overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3, x_1 \vee x_2 \vee x_3, \overline{x}_3\}$. From now on, we only consider the clauses not containing x_1 : $C_1 = \{x_2, \overline{x}_2 \vee x_3, \overline{x}_3\}$, and ignore the clauses containing x_1 : $\{\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3, x_1 \vee x_2 \vee x_3\}$. We continue by resolving the first two clauses of C_1 ; we get $\{x_3, x_2 \vee \overline{x}_3, \overline{x}_3\}$, which is a saturation of C_1 w.r.t. x_2 . Hence, $C_2 = \{x_3, \overline{x}_3\}$ is the resulting multiset of clauses not containing x_2 and ignore $\{x_2 \vee \overline{x}_3\}$, which is the multiset of clauses containing x_2 . Finally, we resolve $\{x_3, \overline{x}_3\}$ and get the empty clause. Since all the variables have been saturated, the minimum number of unsatisfied clauses in ϕ is 1.

The use of restrictions of MaxSAT resolution has not been limited to branch-and-bound solvers. Narodytska and Bacchus [NB14] used MaxSAT resolution in SAT-based MaxSAT solvers, avoiding the use of cardinality constraints and obtaining very competitive results on industrial instances.

There exists no polynomial-size resolution proof of the pigeon hole principle (PHP). However, Ignatiev et al. [IMM17] showed that there exist polynomial-size MaxSAT resolution proofs of PHP if PHP is encoded as a Partial MaxSAT instance using the dual rail encoding. Indeed, the combination of the dual rail encoding and MaxSAT resolution is a stronger proof system than either general resolution or conflict-driven clause learning [BBI⁺18].

MaxSAT resolution has been extended to the multiple-valued clausal forms known as signed CNF formulas [BHM00]. The defined signed MaxSAT resolution rules are complete and provide a logical framework for weighted constraint satisfaction problems (WCSP) [ABLM07b]. Besides, some restrictions of the rules enforce the defined local consistency properties for WCSPs in a natural way [ABLM07a, ABLM13].

23.4.2. Clause MaxSAT Tableaux

The clause SAT tableau calculus [D'A99, Häh01], which is unsound for MaxSAT, was reformulated to become a sound and complete MaxSAT calculus [LMS16b]. Roughly speaking, the main differences are that the extension rule is applied in a branch until all its clauses have been expanded regardless of the contradictions already detected in that branch, and that clauses and literals cannot be used more than once in a branch.

A clause MaxSAT tableau for a multiset of clauses $\phi = \{C_1, \dots, C_m\}$ is a tree with a finite number of branches whose nodes are labelled with clauses that are declared to be either active or inactive in each branch. It is constructed by a sequence of applications of the following rules:

Initialize: A tree with a single branch with m nodes such that each node is labelled with a clause of ϕ is a clause MaxSAT tableau for ϕ . Such a tableau is called initial tableau and its clauses are declared to be active.

Extension: Given a clause MaxSAT tableau T for ϕ , a branch B of T , and a node of B labelled with an active clause $l_1 \vee \dots \vee l_r$ with $r \geq 2$, the tableau obtained by creating r sibling nodes below B and labelling each node with a different unit clause from $\{l_1, \dots, l_r\}$ is a clause MaxSAT tableau for ϕ . Clause $l_1 \vee \dots \vee l_r$ becomes inactive in the new branches, and unit clauses l_1, \dots, l_r are declared to be active.

Contradiction: Given a clause MaxSAT tableau T for ϕ , a branch B of T , and two nodes of B labelled with two active unit clauses l and \bar{l} , the tableau obtained by appending a node labelled with an empty clause below B is a clause MaxSAT tableau for ϕ . The empty clause becomes active and unit clauses l and \bar{l} become inactive.

A clause MaxSAT tableau is completed when all its branches are saturated, in the sense that all its active clauses are empty and unit clauses, and the contradiction rule cannot be further applied in any branch. The cost of a branch in a completed tableau T is the number of empty clauses in it, and the cost of T is the minimum cost among all its branches.

The soundness and completeness of the clause MaxSAT tableau calculus states that the minimum number of clauses that can be unsatisfied in a multiset of clauses ϕ is k iff the cost of a completed clause MaxSAT tableau for ϕ

is k . Thus, the systematic construction of a completed clause MaxSAT tableau for ϕ provides an exact method for MaxSAT, and each completed tableau is a proof.

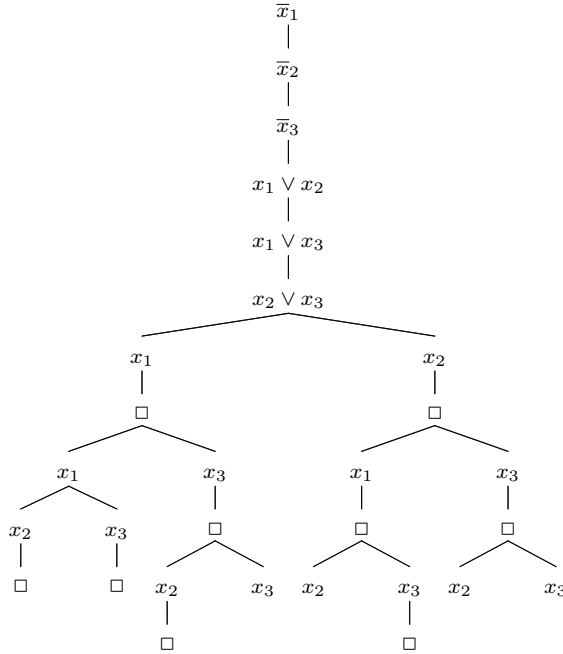


Figure 23.2. A completed clause MaxSAT tableau for $\phi = \{\bar{x}_1, \bar{x}_2, \bar{x}_3, x_1 \vee x_2, x_1 \vee x_3, x_2 \vee x_3\}$.

Figure 23.2 shows a completed clause MaxSAT tableau T for the multiset of clauses $\phi = \{\bar{x}_1, \bar{x}_2, \bar{x}_3, x_1 \vee x_2, x_1 \vee x_3, x_2 \vee x_3\}$. The saturated branches of the tableau have cost 2 except for branches 3 and 6 (counting from left to right) that have cost 3. The active clauses in each branch are: $\{\square, \square, x_1, \bar{x}_3\}$ (branch 1), $\{\square, \square, x_1, \bar{x}_2\}$ (branch 2), $\{\square, \square, \square\}$ (branch 3), $\{\square, \square, \bar{x}_2, x_3\}$ (branch 4), $\{\square, \square, x_2, \bar{x}_3\}$ (branch 5), $\{\square, \square, \square\}$ (branch 6), $\{\square, \square, \bar{x}_1, x_2\}$ (branch 7), and $\{\square, \square, \bar{x}_1, x_3\}$ (branch 8). Therefore, the minimum number of unsatisfied clauses in ϕ is 2.

Inspired on the previous calculus, as well as in the clause MinSAT calculus defined in [LMS16a], Argelich et al. [ALMS18] defined a proof procedure that implements a sound and complete clause tableau-style calculus for both MaxSAT and MinSAT. The expansion rules preserves adequately the number of unsatisfied clauses in the generated subproblems. The leaf nodes of a completed tableau contain a number of empty clauses ranging between the minimum and the maximum number of unsatisfied clauses in the input formula, and there is at least one branch with the minimum value and at least one branch with the maximum value.

Casas-Roma et al. [CHM17] defined a complete natural deduction calculus for MaxSAT which was inspired on the previous clause MaxSAT tableau calculus.

23.5. Approximation Algorithms

Heuristic local search algorithms¹ are often quite effective at finding near-optimal solutions. Actually, most of the exact MaxSAT solvers use a local search algorithm to compute an initial upper bound. However, these algorithms, in contrast with approximation algorithms, do not come with rigorous guarantees concerning the quality of the final solution or the required maximum runtime. Informally, an algorithm approximately solves an optimization problem if it always returns a feasible solution whose measure is close to optimal, for example, within a factor bounded by a constant or by a slowly growing function of the input size. Given a constant α , an algorithm is an α -approximation algorithm for a maximization (minimization) problem if it provides a feasible solution in polynomial time which is at least (most) α times the optimum, considering all the possible instances of the problem.

The first MaxSAT approximation algorithm, with a performance guarantee of $\frac{1}{2}$, is a greedy algorithm that was devised by Johnson in 1974 [Joh74]. This result was improved in 1994 by Yannakakis [Yan94], and Goemans and Williamson [GW94b], who described $\frac{3}{4}$ -approximation algorithms for MaxSAT. Then, Goemans and Williamson [GW94b] proposed a .878-approximation algorithm for Max2SAT (which gives a .7584-approximation for MaxSAT [GW95]) based on semidefinite programming [GW94a]. Since then other improvements have been achieved, but there is a limit on approximability: Hastad [Has97] proved that, unless $P = NP$, no approximation algorithm for MaxSAT (even for Max3SAT) can achieve a performance guarantee better than $\frac{7}{8}$. Interestingly, Karloff and Zwick [KZ97] gave a $\frac{7}{8}$ approximation algorithm for Max3SAT, showing that the constant $\frac{7}{8}$ is tight. The most promising approaches from a theoretical and practical point of view are based on semidefinite programming [GvHL06]. We refer the reader to the survey of Anjos [Anj05] to learn more about how to approximate MaxSAT with semidefinite programming.

23.6. The MaxSAT Evaluation

The MaxSAT Evaluation [ALMP08, ALMP11a, ALMP11b] is an affiliated event of the International Conference on Theory and Applications of Satisfiability Testing. It has been held annually since 2006 and has been decisive for promoting and advancing MaxSAT solving.

The main goals of the MaxSAT Evaluation are to assess the state of the art in the field of MaxSAT solvers, collect and re-distribute a heterogeneous MaxSAT benchmark set for further scientific evaluations, and promote MaxSAT as a viable option for solving instances of a wide range of NP-hard optimization problems. In the beginning, only exact solvers were evaluated. Ultimately, an evaluation of non-exact solvers is also conducted.

Until 2016, the MaxSAT Evaluation had the random, crafted and industrial categories and the unweighted MaxSAT, weighted MaxSAT, partial MaxSAT and weighted partial MaxSAT tracks. Since 2017, it has two tracks: unweighted and

¹We do not include a section about local search and MaxSAT because there is a chapter on local search in the handbook.

weighted. The unweighted track combines the industrial and crafted unweighted and unweighted partial MaxSAT categories from previous MaxSAT evaluations. Purely randomly generated instances are not included. The weighted track combines the industrial and crafted weighted and weighted partial MaxSAT categories from previous MaxSAT evaluations. All benchmarks contain soft clauses with different weights. Purely randomly generated instances are not included.

We refer the reader to <https://maxsat-evaluations.github.io/2018/> for additional information and the results of the last edition of the MaxSAT Evaluation.

23.7. Other Contributions to MaxSAT

First of all, we should mention a variety of applications of MaxSAT in a range of real-world domains as diverse as bioinformatics [GL12, MAGL11], circuit design and debugging [SMV⁺07], community detection in complex networks [JMRS17], diagnosis [DG12], FPGA routing [XRS03], planning [ZB12], scheduling [BGSV15], team formation [MNRS17] and time tabling [AN14], among many others.

Other research topics that have appeared in the literature and contain substantial contributions to the field of MaxSAT solving include the definition of clause clause learning schemes in branch-and-bound solvers [AH16, AM06b] that are not yet competitive enough to solve industrial instances, the creation of robust MaxSAT solutions [BBMV13], the definition of efficient encodings from MaxCSP to MaxSAT [ACLM12], the extension of MaxSAT to many-valued logic [ALM17] and the definition of MaxSAT formalisms that deal with blocks of soft clauses instead of individual soft clauses [AM06a, HMM15].

23.8. The MinSAT Problem

Given the success of MaxSAT, the community has started to look into MinSAT. At first sight, one could think that the solving techniques and encodings to be used in MinSAT are very similar to the ones used in MaxSAT and, therefore, that there is no need of investigating MinSAT from a problem solving perspective. However, most of the research conducted so far indicates that they may be quite different, as well as that the performance profile of MaxSAT and MinSAT is also different for several optimization problems represented into these formalisms [ALMZ14, IMM14, LZMS12]. It is also worth mentioning that MinSAT is meaningful for both satisfiable and unsatisfiable instances, whereas MaxSAT is only meaningful for unsatisfiable instances. A closely related problem has been analyzed in [IMPMS13, IMPM16].

The work on MinSAT for solving optimization problems may be divided into the following categories:

- Transformations between MinSAT and MaxSAT: Reductions from MinSAT to partial MaxSAT were defined in [LMQZ10], but these reductions do not generalize to weighted partial MinSAT. This drawback was overcome with the definition of the natural encoding [K12], which was improved in [ZLMA12]. Reductions of weighted partial MinSAT to Group MaxSAT were evaluated in [HMPMS12].

- Encodings from weighted MaxCSP to MinSAT: Efficient encodings were defined in [ALMZ13]. Using the MaxSAT direct encoding [ACLM12], we must add one clause for every no-good, while using the MinSAT direct encoding [ALMZ13], we must instead add one clause for every good. This implies, for instance, that for representing the constraint $X = Y$, we need a number of clauses linear in the domain size in MinSAT, and a quadratic number of clauses in MaxSAT. We are in the opposite situation if we want to represent the constraint $X \neq Y$. So, it seems that MaxSAT and MinSAT could be complementary in some scenarios [ALMZ13].
- Complete logical calculi: MaxSAT resolution is sound for MinSAT but to get completeness the elimination of variables must be defined differently [LM15]. After saturating a variable x , the clauses containing the variable x are ignored in MaxSAT but, in MinSAT, the resulting clauses of eliminating the occurrences of both x and \bar{x} must also be considered in the saturation of the next variable. In this way, after saturating all the variables, the number of empty clauses derived is equal to the maximum number of clauses that can be unsatisfied. In the case of clause tableaux, the clause MaxSAT calculus is unsound for MinSAT because the extension rule does not preserve the maximum number of unsatisfied clauses. A sound and complete clause MinSAT calculus was defined in [LMS16a].
- MinSAT solvers: The only existing branch-and-bound MinSAT solver, MinSatz [LZMS11, LZMS12], is based on MaxSatz and implements upper bounds that exploit clique partition algorithms and MaxSAT technology. There exist two SAT-based MinSAT solvers of this class [ALMZ12, HMPMS12]. They differ with SAT-based MaxSAT solvers in the way of relaxing soft clauses. A local search MinSAT solver was described in [AH15a].

23.9. Conclusions

We have presented an overview about MaxSAT and focused on the solving techniques that have proved to be useful in terms of performance. When we wrote the chapter on MaxSAT for the first edition of this handbook, MaxSAT solving was still an incipient research topic but nowadays can be considered a mature research topic with a variety of applications and also of open problems.

From the perspective of branch-and-bound MaxSAT solvers, the main challenge is to build a solver as competitive as SAT-based MaxSAT solvers on industrial instances. A first step in this direction could be the definition of a powerful clause learning mechanism for soft clauses. From a logical perspective, an interesting research avenue is the study of the proof complexity of logical calculi for MaxSAT to find proof systems more efficient than the proof systems used in SAT solving. From a knowledge representation perspective, an interesting research path is to find clever encodings of combinatorial optimization problems and understand the impact of modeling on the performance of MaxSAT solvers, as well as the definition of richer formalisms like non-clausal MaxSAT.

Acknowledgments

This work was supported by Project LOGISTAR from the EU H2020 Research and Innovation Programme under Grant Agreement No. 769142, and MINECO-FEDER project RASO (TIN2015-71799-C2-1-P).

References

- [ABLM07a] C. Ansótegui, M. L. Bonet, J. Levy, and F. Manyà. Inference rules for high-order consistency in weighted CSP. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence, Vancouver, Canada*, pages 167–172, 2007.
- [ABLM07b] C. Ansótegui, M. L. Bonet, J. Levy, and F. Manyà. The logic behind weighted CSP. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI-2007, Hyderabad, India*, pages 32–37, 2007.
- [ABLM13] C. Ansótegui, M. L. Bonet, J. Levy, and F. Manyà. Resolution procedures for multiple-valued optimization. *Information Sciences*, 227:43–59, 2013.
- [ACLM12] J. Argelich, A. Cabiscol, I. Lynce, and F. Manyà. Efficient encodings from CSP into SAT, and from MaxCSP into MaxSAT. *Multiple-Valued Logic and Soft Computing*, 19(1-3):3–23, 2012.
- [AG13] C. Ansótegui and J. Gabàs. Solving (weighted) partial MaxSAT with ILP. In *Proceedings of the 10th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR 2013, Yorktown Heights, NY, USA*, pages 403–409, 2013.
- [AH14a] A. Abramé and D. Habet. Ahmaxsat: Description and evaluation of a branch and bound Max-SAT solver. *JSAT*, 9:89–128, 2014.
- [AH14b] A. Abramé and D. Habet. Local max-resolution in branch and bound solvers for Max-SAT. In *Proceedings of the 26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI, Limassol, Cyprus*, pages 336–343, 2014.
- [AH14c] A. Abramé and D. Habet. Maintaining and handling all unit propagation reasons in exact Max-SAT solvers. In *Proceedings of the Seventh Annual Symposium on Combinatorial Search, SOCS, Prague, Czech Republic*, 2014.
- [AH15a] A. Abramé and D. Habet. Local search algorithm for the partial minimum satisfiability problem. In *Proceedings of the 27th IEEE International Conference on Tools with Artificial Intelligence, ICTAI, Vietri sul Mare, Italy*, pages 821–827, 2015.
- [AH15b] A. Abramé and D. Habet. On the resiliency of unit propagation to Max-Resolution. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence, IJCAI, Buenos Aires, Argentina*, pages 268–274, 2015.
- [AH16] A. Abramé and D. Habet. Learning nobetter clauses in Max-SAT branch and bound solvers. In *Proceedings of the 28th IEEE Interna-*

tional Conference on Tools with Artificial Intelligence, ICTAI, San Jose, CA, USA, pages 452–459, 2016.

- [ALM17] J. Argelich, C. M. Li, and F. Manyà. Exploiting many-valued variables in MaxSAT. In *Proceedings of the 7th IEEE International Symposium on Multiple-Valued Logic, ISMVL, Novi Sad, Serbia*, pages 155–160, 2017.
- [ALMP08] J. Argelich, C. M. Li, F. Manyà, and J. Planes. The first and second Max-SAT evaluations. *Journal on Satisfiability, Boolean Modeling and Computation*, 4, 2008.
- [ALMP11a] J. Argelich, C. M. Li, F. Manyà, and J. Planes. Analyzing the instances of the maxsat evaluation. In *Proceedings of the 14th International Conference on Theory and Applications of Satisfiability Testing, SAT-2011, Ann Arbor, MI, USA*, pages 360–361. Springer LNCS 6695, 2011.
- [ALMP11b] J. Argelich, C. M. Li, F. Manyà, and J. Planes. Experimenting with the instances of the MaxSAT evaluation. In *Proceedings of the 14th International Conference of the Catalan Association for Artificial Intelligence, CCIA-2011, Lleida, Spain*, pages 31–40. IOS Press, 2011.
- [ALMS18] J. Argelich, C. M. Li, F. Manyà, and J. R. Soler. Clause branching in MaxSAT and MinSAT. In *Proceedings of the 21st International Conference of the Catalan Association for Artificial Intelligence, Roses, Spain*, volume 308 of *Frontiers in Artificial Intelligence and Applications*, pages 17–26. IOS Press, 2018.
- [ALMZ12] C. Ansótegui, C. M. Li, F. Manyà, and Z. Zhu. A SAT-based approach to MinSAT. In *Proceedings of the 15th International Conference of the Catalan Association for Artificial Intelligence, CCIA-2012, Alacant, Spain*, pages 185–189. IOS Press, 2012.
- [ALMZ13] J. Argelich, C. M. Li, F. Manyà, and Z. Zhu. MinSAT versus MaxSAT for optimization problems. In *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming, CP 2013, Uppsala, Sweden*, pages 133–142. Springer LNCS 8124, 2013.
- [ALMZ14] J. Argelich, C. M. Li, F. Manyà, and Z. Zhu. Many-valued MinSAT solving. In *Proceedings, 44th International Symposium on Multiple-Valued Logics (ISMVL), Bremen, Germany*, pages 32–37. IEEE CS Press, 2014.
- [AM06a] J. Argelich and F. Manyà. Exact Max-SAT solvers for over-constrained problems. *Journal of Heuristics*, 12(4–5):375–392, 2006.
- [AM06b] J. Argelich and F. Manyà. Learning hard constraints in Max-SAT. In *Proceedings of the Workshop on Constraint Solving and Constraint Logic Programming, CSCLP-2006, Lisbon, Portugal*, pages 1–12, 2006.
- [AMP03] T. Alsinet, F. Manyà, and J. Planes. Improved branch and bound algorithms for Max-SAT. In *Proceedings of the 6th International Conference on the Theory and Applications of Satisfiability Testing*, 2003.

- [AMP04] T. Alsinet, F. Manyà, and J. Planes. A Max-SAT solver with lazy data structures. In *Proceedings of the 9th Ibero-American Conference on Artificial Intelligence, IBERAMIA 2004, Puebla, México*, pages 334–342. Springer LNCS 3315, 2004.
- [AMP05] T. Alsinet, F. Manyà, and J. Planes. Improved exact solver for weighted Max-SAT. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing, SAT-2005, St. Andrews, Scotland*, pages 371–377. Springer LNCS 3569, 2005.
- [AMP08] T. Alsinet, F. Manyà, and J. Planes. An efficient solver for Weighted Max-SAT. *Journal of Global Optimization*, 41:61–73, 2008.
- [AN14] R. J. A. Achá and R. Nieuwenhuis. Curriculum-based course timetabling with SAT and MaxSAT. *Annals OR*, 218(1):71–91, 2014.
- [Anj05] M. F. Anjos. Semidefinite optimization approaches for satisfiability and maximum-satisfiability problems. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:1–47, 2005.
- [BBI⁺18] M. L. Bonet, S. Buss, A. Ignatiev, J. Marques-Silva, and A. Morgado. MaxSAT resolution with the dual rail encoding. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence, AAAI, New Orleans, Louisiana, USA*, pages 6565–6572, 2018.
- [BBMV13] M. Boffill, D. Busquets, V. Muñoz, and M. Villaret. Reformulation based MaxSAT robustness. *Constraints*, 18(2):202–235, 2013.
- [BF99] B. Borchers and J. Furman. A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems. *Journal of Combinatorial Optimization*, 2:299–306, 1999.
- [BGSV15] M. Boffill, M. Garcia, J. Suy, and M. Villaret. MaxSAT-based scheduling of B2B meetings. In *Proceedings of the 12th International Conference on Integration of AI and OR Techniques in Constraint Programming, CPAIOR, Barcelona, Spain*, pages 65–73, 2015.
- [BHM00] B. Beckert, R. Hähnle, and F. Manyà. The SAT problem of signed CNF formulas. In D. Basin, M. D’Agostino, D. Gabbay, S. Matthews, and L. Viganò, editors, *Labelled Deduction*, volume 17 of *Applied Logic Series*, pages 61–82. Kluwer, Dordrecht, 2000.
- [BLM06] M. L. Bonet, J. Levy, and F. Manyà. A complete calculus for Max-SAT. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing, SAT-2006, Seattle, USA*, pages 240–251. Springer LNCS 4121, 2006.
- [BLM07] M. L. Bonet, J. Levy, and F. Manyà. Resolution for Max-SAT. *Artificial Intelligence*, 171(8–9):240–251, 2007.
- [BR99] N. Bansal and V. Raman. Upper bounds for MaxSat: Further improved. In *Proc 10th International Symposium on Algorithms and Computation, ISAAC’99, Chennai, India*, pages 247–260. Springer, LNCS 1741, 1999.
- [CHM17] J. Casas-Roma, A. Huertas, and F. Manyà. Solving MaxSAT with natural deduction. In *Proceedings of the 20th International Conference of the Catalan Association for Artificial Intelligence, Deltebre, Spain*, volume 300 of *Frontiers in Artificial Intelligence and Appli-*

- cations*, pages 186–195. IOS Press, 2017.
- [D’A99] M. D’Agostino. Tableaux methods for classical propositional logic. In M. D’Agostino, D. Gabbay, R. Hähnle, and J. Posegga, editors, *Handbook of Tableau Methods*, pages 45–123. Kluwer, 1999.
- [DDDL07] S. Darras, G. Dequen, L. Devendeville, and C. M. Li. On inconsistent clause-subsets for max-sat solving. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming, CP-2007, Providence/RI, USA*, pages 225–240. Springer LNCS 4741, 2007.
- [DG12] D. D’Almeida and É. Grégoire. Model-based diagnosis with default information implemented through MAX-SAT technology. In *Proceedings of the IEEE 13th International Conference on Information Reuse & Integration, IRI, Las Vegas, NV, USA*, pages 33–36, 2012.
- [dGLMS03] S. de Givry, J. Larrosa, P. Meseguer, and T. Schiex. Solving MaxSAT as weighted CSP. In *9th International Conference on Principles and Practice of Constraint Programming, CP-2003, Kinsale, Ireland*, pages 363–376. Springer LNCS 2833, 2003.
- [GL12] J. Guerra and I. Lynce. Reasoning over biological networks using maximum satisfiability. In *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming, CP, Québec City, QC, Canada*, pages 941–956, 2012.
- [GvHL06] C. P. Gomes, W.-J. van Hove, and L. Leahu. The power of semidefinite programming relaxations for MAXSAT. In *Proceedings of the Third International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR06, Cork, Ireland*, pages 104–118. Springer LNCS 3990, 2006.
- [GW94a] M. X. Goemans and D. P. Williamson. .878-approximation algorithms for MAX CUT and MAX 2SAT. In *Proceedings of the 26th ACM Symposium on the Theory of Computing*, pages 422–431, 1994.
- [GW94b] M. X. Goemans and D. P. Williamson. New 3/4-approximation algorithms for the maximum satisfiability problem. *SIAM Journal of Discrete Mathematics*, 7:656–666, 1994.
- [GW95] M. X. Goemans and D. P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42:1115–1145, 1995.
- [Häh01] R. Hähnle. Tableaux and related methods. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 100–178. Elsevier and MIT Press, 2001.
- [Has97] J. Hastad. Some optimal inapproximability results. In *Proceedings of the 28th ACM Symposium on the Theory of Computing*, pages 1–10, 1997.
- [HL06] F. Heras and J. Larrosa. New inference rules for efficient MaxSAT solving. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-2006, Boston/MA, USA*, pages 68–73, 2006.
- [HLO07] F. Heras, J. Larrosa, and A. Oliveras. MiniMaxSat: A new weighted Max-SAT solver. In *Proceedings of the 10th International Confer-*

- ence on Theory and Applications of Satisfiability Testing, SAT-2007, Lisbon, Portugal, pages 41–55, 2007.
- [HLO08] F. Heras, J. Larrosa, and A. Oliveras. MiniMaxSAT: An efficient weighted Max-SAT solver. *Journal of Artificial Intelligence Research*, 31:1–32, 2008.
- [HMM15] F. Heras, A. Morgado, and J. Marques-Silva. MaxSAT-based encodings for group MaxSAT. *AI Communications*, 28(2):195–214, 2015.
- [HMPMS12] F. Heras, A. Morgado, J. Planes, and J. Marques-Silva. Iterative SAT solving for minimum satisfiability. In *Proceedings of the IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012, Athens, Greece*, pages 922–927, 2012.
- [IMM14] A. Ignatiev, A. Morgado, and J. Marques-Silva. On reducing maximum independent set to minimum satisfiability. In *Proc 17th International Conference on Theory and Applications of Satisfiability Testing, SAT, Vienna, Austria*, pages 103–120. Springer, LNAI 8561, 2014.
- [IMM17] A. Ignatiev, A. Morgado, and J. Marques-Silva. On tackling the limits of resolution in SAT solving. In *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing, SAT, Melbourne, Australia*, pages 164–183. Springer LNCS 10491, 2017.
- [IMPM16] A. Ignatiev, A. Morgado, J. Planes, and J. Marques-Silva. Maximal falsifiability. *AI Communications*, 29(2):351–370, 2016.
- [IMPMS13] A. Ignatiev, A. Morgado, J. Planes, and J. Marques-Silva. Maximal falsifiability - definitions, algorithms, and applications. In *Proc 19th International Conference on Logic Programming and Automated Theorem Proving, LPAR, Stellenbosch, South Africa*, pages 439–456. Springer, LNAI 8312, 2013.
- [JLM17] H. Jiang, C. M. Li, and F. Manyà. An exact algorithm for the maximum weight clique problem of large graphs. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence, AAAI, San Francisco/CA, USA*, 2017.
- [JMRS17] S. Jabbour, N. Mhadhbi, B. Raddaoui, and L. Sais. A SAT-based framework for overlapping community detection in networks. In *Proceedings of the 21st Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining, Part II, PAKDD, Jeju, South Korea*, pages 786–798, 2017.
- [Joh74] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.
- [K12] A. Kügel. Natural Max-SAT encoding of Min-SAT. In *Proceedings of the Learning and Intelligent Optimization Conference, LION 6, Paris, France*, 2012.
- [Kue10] A. Kuegel. Improved exact solver for the Weighted MAX-SAT problem. In *Proceedings of Workshop Pragmatics of SAT, POS-10, Edinburgh, UK*, pages 15–27, 2010.
- [KZ97] H. Karloff and U. Zwick. A 7/8-approximation algorithm for

- MAX3SAT? In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science, FOCS'97*, pages 406–415, 1997.
- [LFX13] C. Li, Z. Fang, and K. Xu. Combining MaxSAT reasoning and incremental upper bound for the maximum clique problem. In *Proceedings of the 25th IEEE International Conference on Tools with Artificial Intelligence, ICTAI, Herndon, VA, USA*, pages 939–946, 2013.
- [LH05] J. Larrosa and F. Heras. Resolution in Max-SAT and its relation to local consistency in weighted CSPs. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI-2005, Edinburgh, Scotland*, pages 193–198. Morgan Kaufmann, 2005.
- [LHdG08] J. Larrosa, F. Heras, and S. de Givry. A logical approach to efficient Max-SAT solving. *Artificial Intelligence*, 172(2–3):204–233, 2008.
- [LJM17] C. Li, H. Jiang, and F. Manyà. On minimization of the number of branches in branch-and-bound algorithms for the maximum clique problem. *Computers & OR*, 84:1–15, 2017.
- [LJX15] C. Li, H. Jiang, and R. Xu. Incremental maxsat reasoning to reduce branches in a branch-and-bound algorithm for maxclique. In *Proceedings of the 9th International Conference on Learning and Intelligent Optimization, LION, Lille, France*, pages 268–274, 2015.
- [LM15] C. M. Li and F. Manyà. An exact inference scheme for MinSAT. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence, IJCAI-2015, Buenos Aires, Argentina*, pages 1959–1965, 2015.
- [LMMP08] C. M. Li, F. Manyà, N. O. Mohamedou, and J. Planes. Transforming inconsistent subformulas in MaxSAT lower bound computation. In *Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming, CP-2008, Sydney, Australia*. Springer LNCS, 2008.
- [LMMP09] C. M. Li, F. Manyà, N. O. Mohamedou, and J. Planes. Exploiting cycle structures in Max-SAT. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing, SAT-2009, Swansea, UK*, pages 467–480. Springer LNCS 5584, 2009.
- [LMMP10] C. M. Li, F. Manyà, N. O. Mohamedou, and J. Planes. Resolution-based lower bounds in MaxSAT. *Constraints*, 15(4):456–484, 2010.
- [LMP05] C. M. Li, F. Manyà, and J. Planes. Exploiting unit propagation to compute lower bounds in branch and bound Max-SAT solvers. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming, CP-2005, Sitges, Spain*, pages 403–414. Springer LNCS 3709, 2005.
- [LMP06] C. M. Li, F. Manyà, and J. Planes. Detecting disjoint inconsistent subformulas for computing lower bounds for Max-SAT. In *Proceedings of the 21st National Conference on Artificial Intelligence, AAAI-2006, Boston/MA, USA*, pages 86–91, 2006.
- [LMP07] C. M. Li, F. Manyà, and J. Planes. New inference rules for Max-SAT. *Journal of Artificial Intelligence Research*, 30:321–359, 2007.

- [LMQZ10] C. M. Li, F. Manyà, Z. Quan, and Z. Zhu. Exact MinSAT solving. In *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing, SAT-2010, Edinburgh, UK*, pages 363–368. Springer LNCS 6175, 2010.
- [LMS16a] C. M. Li, F. Manyà, and J. R. Soler. A clause tableau calculus for MinSAT. In *Proceedings of the 19th International Conference of the Catalan Association for Artificial Intelligence, C CIA 2016, Barcelona, Spain*, volume 288 of *Frontiers in Artificial Intelligence and Applications*, pages 88–97. IOS Press, 2016.
- [LMS16b] C. M. Li, F. Manyà, and J. R. Soler. A clause tableaux calculus for MaxSAT. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence, IJCAI-2016, New York, USA*, pages 766–772, 2016.
- [LQ10] C. M. Li and Z. Quan. An efficient branch-and-bound algorithm based on MaxSAT for the maximum clique problem. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence, AAAI, Atlanta, Georgia, USA*, 2010.
- [LS07] H. Lin and K. Su. Exploiting inference rules to compute lower bounds for MAX-SAT solving. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI-2007, Hyderabad, India*, pages 2334–2339, 2007.
- [LSL08] H. Lin, K. Su, and C. M. Li. Within-problem learning for efficient lower bound computation in Max-SAT solving. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence, AAAI-2008, Chicago/IL, USA*, pages 351–356, 2008.
- [LZMS11] C. M. Li, Z. Zhu, F. Manyà, and L. Simon. Minimum satisfiability and its applications. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence, IJCAI-2011, Barcelona, Spain*, pages 605–610, 2011.
- [LZMS12] C. M. Li, Z. Zhu, F. Manyà, and L. Simon. Optimizing with minimum satisfiability. *Artificial Intelligence*, 190:32–44, 2012.
- [MAGL11] J. Marques-Silva, J. Argelich, A. Graça, and I. Lynce. Boolean lexicographic optimization: algorithms & applications. *Annals of Mathematics and Artificial Intelligence*, 62(3-4):317–343, 2011.
- [MNRS17] F. Manyà, S. Negrete, C. Roig, and J. R. Soler. A MaxSAT-based approach to the team composition problem in a classroom. In *Autonomous Agents and Multiagent Systems - AAMAS 2017 Workshops, Visionary Papers, São Paulo, Brazil, Revised Selected Papers*, pages 164–173. Springer LNCS 10643, 2017.
- [NB14] N. Narodytska and F. Bacchus. Maximum satisfiability using core-guided MaxSAT resolution. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence, Québec City, Canada.*, pages 2717–2723, 2014.
- [NR00] R. Niedermeier and P. Rossmanith. New upper bounds for maximum satisfiability. *Journal of Algorithms*, 36:63–88, 2000.
- [PD07] K. Pipatsrisawat and A. Darwiche. Clone: Solving weighted MaxSAT in a reduced search space. In *20th Australian Joint Conference*

- on *Artificial Intelligence, AI-07, Queensland, Australia*, pages 223–233, 2007.
- [PPC⁺08] K. Pipatsrisawat, A. Palyan, M. Chavira, A. Choi, and A. Darwiche. Solving weighted Max-SAT problems in a reduced search space: A performance analysis. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:191–217, 2008.
- [RG07] M. Ramírez and H. Geffner. Structural relaxations by variable renaming and their compilation for solving MinCostSAT. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming, CP-2007, Providence/RI, USA*, pages 605–619. Springer LNCS 4741, 2007.
- [SMV⁺07] S. Safarpour, H. Mangassarian, A. G. Veneris, M. H. Liffiton, and K. A. Sakallah. Improved design debugging using maximum satisfiability. In *Proceedings of 7th International Conference on Formal Methods in Computer-Aided Design, FMCAD, Austin, Texas, USA*, pages 13–19, 2007.
- [SZ04] H. Shen and H. Zhang. Study of lower bound functions for max-2-sat. In *Proceedings of the 19th National Conference on Artificial Intelligence, AAAI-2004, San Jose/CA, USA*, pages 185–190, 2004.
- [WF96] R. J. Wallace and E. Freuder. Comparative studies of constraint satisfaction and Davis-Putnam algorithms for maximum satisfiability problems. In D. Johnson and M. Trick, editors, *Cliques, Coloring and Satisfiability*, volume 26, pages 587–615. American Mathematical Society, 1996.
- [XRS03] H. Xu, R. A. Rutenbar, and K. A. Sakallah. sub-sat: a formulation for relaxed boolean satisfiability with applications in routing. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 22(6):814–820, 2003.
- [XZ04] Z. Xing and W. Zhang. Efficient strategies for (weighted) maximum satisfiability. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming, CP-2004, Toronto, Canada*, pages 690–705. Springer, LNCS 3258, 2004.
- [XZ05] Z. Xing and W. Zhang. An efficient exact algorithm for (weighted) maximum satisfiability. *Artificial Intelligence*, 164(2):47–80, 2005.
- [Yan94] M. Yannakakis. On the approximation of maximum satisfiability. *Journal of Algorithms*, 17:475–502, 1994.
- [ZB12] L. Zhang and F. Bacchus. MAXSAT heuristics for cost optimal planning. In *Proceedings of the 26th AAAI Conference on Artificial Intelligence, Toronto, Ontario, Canada*, pages 1846–1852, 2012.
- [ZLMA12] Z. Zhu, C. M. Li, F. Manyà, and J. Argelich. A new encoding from MinSAT into MaxSAT. In *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming, CP 2012, Québec City, QC, Canada*, pages 455–463. Springer LNCS 7514, 2012.
- [ZSM03] H. Zhang, H. Shen, and F. Manyà. Exact algorithms for MAX-SAT. *Electronic Notes in Theoretical Computer Science*, 86(1), 2003.

This page intentionally left blank

Chapter 24

Maximum Satisfiability

Fahiem Bacchus, Matti Järvisalo, and Ruben Martins

Maximum satisfiability (MaxSAT) is an optimization version of SAT that is solved by finding an optimal truth assignment instead of just a satisfying one. In MaxSAT the objective function to be optimized is specified by a set of weighted soft clauses: the objective value of a truth assignment is the sum of the weights of the soft clauses it satisfies. In addition, the MaxSAT problem can have hard clauses that the truth assignment must satisfy. Many optimization problems can be naturally encoded into MaxSAT and this, along with significant performance improvements in MaxSAT solvers, has led to MaxSAT being used in a number of different application areas. This chapter provides a detailed overview of the approaches to MaxSAT solving that have in recent years been most successful in solving real-world optimization problems. Further recent developments in MaxSAT research are also overviewed, including encodings, applications, preprocessing, incomplete solving, algorithm portfolios, partitioning-based solving, and parallel solving.

24.1. Introduction

Computational problems that arise from real-world application scenarios often involve constraints that specify the feasible solutions and some notion of cost or goodness associated with the different solutions. Whereas satisfiability (SAT) provides no direct way of distinguishing solutions in terms of their cost or goodness, maximum satisfiability (MaxSAT for short) generalizes SAT with a over the solutions. This cost function is specified by a set of soft clauses with weights: the cost of a solution is the sum of the weights of the soft clauses it fails to satisfy. (Equivalently, the objective value of a solution is the sum of the weights of the soft clauses it satisfies.)

MaxSAT was for most of its history used mainly as a theoretical problem in the analysis of the computational complexity of optimization problems. Within the last decade, however, with the rise of increasingly effective MaxSAT solving techniques and solver implementations, MaxSAT has changed from a theoretical formalism into a viable constraint optimization paradigm. The approach of encoding various NP-hard optimization problems into MaxSAT and then utilizing

MaxSAT solvers has proved to be a viable solution technique capable of efficiently solving problems from a variety of domains. These domains include combinatorics, planning and scheduling, verification, security and software management, data analysis and machine learning, knowledge representation, and bioinformatics, among others. Many problems from these domains have natural propositional encodings on which MaxSAT is particularly effective, often more effective than other constraint optimization paradigms such as integer programming (see, e.g., [BJ17] and the results of [BJM14] in comparison with those of [PFL14]). Section 24.3.2 discusses some of these applications of MaxSAT and provides references to the relevant literature.

A series of MaxSAT evaluations [ALMP08, HLG08, ALMP11, ABJM17, BJM18, BJM19], running yearly since 2006 has supported the development of increasingly robust and efficient MaxSAT solvers containing new algorithmic insights. The evaluations provide a yearly snapshot of the current state of the art in MaxSAT solving, and also provide standard benchmark sets for the scientific evaluation of new algorithmic ideas for MaxSAT. The evaluation series is focused on complete MaxSAT solving, i.e., approaches that provide optimal solutions. Recently, however, the evaluations have also added a track for incomplete solvers, with the motivation of developing approaches which can rapidly find good—though not necessarily optimal—solutions.

Complementing Chapter 23, where various branch-and-bound approaches to MaxSAT solving are described (e.g., [HLO08, LMMP10]), the focus of this chapter is on recent algorithmic advances in MaxSAT solving. In particular, we overview MaxSAT solving approaches that utilize iterative SAT solver calls, namely, the model-improving, the core-guided, and the implicit hitting set approaches to MaxSAT solving, which at the time this chapter was written constitute the main algorithmic approaches for solving large MaxSAT instances arising from real-world domains.

The model-improving approach is based on querying a SAT solver for a solution of lower cost until the solution cost can no longer be improved. Cardinality constraints (Section 24.4.2.1) are used to force the SAT solver to find a lower cost solution. The core-guided and implicit hitting set approaches iteratively query a SAT solver for unsatisfiable cores, which are subsets of soft clauses of which at least one must be falsified by any solution. In the core-guided approaches, these cores are used to construct cardinality constraints that allow the SAT solver to falsify one, but no more than one, soft clause from each core. If the problem remains unsatisfiable, a new core will be found and in the next call the SAT solver will be additionally allowed to falsify one, but no more than one, soft clause of this new core. Eventually, no more cores will be found, i.e., the problem will become satisfiable and the solution returned by the SAT solver will falsify one soft clause from each core. Since every solution must falsify at least one soft clause from every core the returned solution will be optimal.

In the implicit hitting set approach cardinality constraints are not used, which in practice makes the SAT solver calls faster. Rather a minimum-cost hitting set of the accumulated set of cores is computed. The cost of this hitting set provides a lower bound on the cost of optimal solutions, as every solution must incur the cost of at least one soft clause from every core (i.e., it must falsify at least one soft

clause from every core). The SAT solver is then asked to find a solution satisfying all of the soft clauses not in the computed hitting set. If it succeeds that solution must be optimal as it will have a cost equal to the lower bound; and if it does not it will find another core, augmenting the set of cores, and a new hitting set can be computed.

These two approaches rely on assumption-based SAT solving to extract cores when no satisfying solution can be found. Assumption-based SAT solving will be described in Section 24.4.1.2; for further details various aspects of modern SAT solvers we refer the interested reader to Chapter 4.

Beyond overviewing these recent algorithmic advances, we also overview some of the recent successful applications of MaxSAT solving, and briefly discuss some further advances in the field, including preprocessing, parallel solving, and incomplete solving.

In the rest of this chapter, we first formalize the MaxSAT problem. Then using this formalism we discuss some encoding techniques that can be used when one wants to encode different problems into MaxSAT, and give an overview of a range of different application problems that have been solved by encoding to MaxSAT and using MaxSAT solvers (Section 24.3). With this motivation of the usefulness of MaxSAT, we turn to the bulk of the chapter: a presentation of recent algorithmic techniques for solving MaxSAT, preceded by some additional needed background (Section 24.4). Finally, we close the chapter with a discussion of other recent developments (Section 24.5) and a summary and outlook on MaxSAT.

24.2. The MaxSAT Formalism

As with SAT, in MaxSAT we deal exclusively with propositional formulas expressed in conjunctive normal form (CNF): each propositional variable and its negation are **literals**; a clause is a disjunction of literals; and a CNF formula is a conjunction of clauses. Since duplicated disjuncts and conjuncts can be discarded a clause can also be viewed as being a set of literals, and a CNF formula as being a set of clauses.

Given a CNF formula F we use $\text{var}(F)$ to denote the set of propositional variables appearing in F . A truth assignment π for F assigns to a truth value (*true* or *false*) to each propositional variable $p \in \text{var}(F)$. For a literal l , π makes l *true* ($\pi \models l$) if l is the variable p and $\pi(p) = \text{true}$ or if l is $\neg p$ and $\pi(p) = \text{false}$. For a clause c , π satisfies c ($\pi \models c$) if π makes at least one of the literals l in c *true*; and π **satisfies** F (or is a model of F) ($\pi \models F$), if it satisfies every clause of F . Given a set of propositional variables $A \subseteq \text{var}(F)$, and a truth assignment π for F we denote the **restriction** of π to A by $\pi|_A$; $\pi|_A$ is a new truth assignment identical to π but only assigning truth values to the variables of A .

A MaxSAT formula F is a CNF formula that is partitioned into hard and soft clauses: $F = \text{hard}(F) \cup \text{soft}(F)$. The hard clauses must be satisfied, while the soft clauses can be falsified at a cost. In particular, each soft clause $c \in \text{soft}(F)$ has an associated positive integral weight $w(c)$ specifying the cost of falsifying that clause.¹ For example, $F = \{(x, y), (\neg x, r, z), (\neg x)_2, (\neg y)_5\}$ is a

¹It is possible to generalize the framework to accommodate rational valued weights, and

MaxSAT formula where the first two clauses are hard and the third and fourth clauses are soft with weights 2 and 5 respectively. We will use subscripted clauses to indicate that they are soft, where the subscript is the soft clause's weight.

A **feasible solution** of F is a truth assignment to the variables of F , $v r (F)$, that satisfies $h rd(F)$. The cost of a feasible solution π of F is the sum of the weights of the soft clauses it falsifies: $o (\pi, F) = \sum_{\{c|c \in soft(F) \wedge \pi \not\models c\}} w (c)$. When F is clear from the context we often write just $o (\pi)$ to denote $o (\pi, F)$.

An **optimal solution** π of F is a feasible solution with minimum cost: $o (\pi, F) \leq o (\pi', F)$ for all feasible solutions π' of F . An optimal solution could equivalently be defined as a feasible solution *satisfying a maximum weight of soft clauses*. We will also use $o (F)$ to denote the cost of an optimal solution of F , and if H is a set of soft clauses $o (H)$ to denote the sum of the weights of its clauses $\sum_{c \in H} w (c)$.

Definition 1 (The MaxSAT problem). Given a MaxSAT formula F find one of its optimal solutions.

Note that F has feasible solutions if and only if $h rd(F)$ is satisfiable. Thus F will have some optimal solutions if and only if $h rd(F)$ is satisfiable. *In the rest of this chapter, we will assume that $h rd(F)$ is satisfiable.* In practice, one can check the satisfiability of $h rd(F)$ with a SAT solver prior to searching for an optimal solution. In fact, such a check is the first step of many MaxSAT algorithms.

Another useful concept is that of a **core**. In SAT, a core is defined to be an unsatisfiable subset of the input formula. In MaxSAT, since the hard clauses must always be satisfied, it is more useful to define a core as follows.

Definition 2 (MaxSAT core). Given a MaxSAT formula F , any subset K of $o (F)$ such that $K \cup h rd(F)$ is unsatisfiable, is a **core** of F . That is, K is a core if and only if every feasible solution of F falsifies at least one soft clause of K .

Historically, restricted forms of MaxSAT have been defined. In particular, a *partial* MaxSAT formula F is one in which $h rd(F)$ is non-empty and a *unweighted* MaxSAT formula F is one in which all the soft clauses have the same weight (which, without loss of generality, can be assumed to be one). Thus four different types of MaxSAT formulas were traditionally recognized: (1) unweighted MaxSAT, (2) weighted MaxSAT, (3) unweighted partial MaxSAT, and (4) weighted partial MaxSAT. The definition given above corresponds to the most general case (4). In recent years the importance of these distinctions has faded as most MaxSAT solvers can handle the most general input. It is still the case, however, that whether or not the instance is weighted or unweighted has an impact on which algorithms perform better. Hence, the yearly MaxSAT evaluations use different tracks for weighted and unweighted instances.

some MaxSAT solvers (e.g., the implicit hitting set solvers described in Section 24.4.6) can solve problems containing soft clauses with floating point weights.

24.3. Encodings and Applications

Significant performance improvements in MaxSAT algorithms over the years have led to an increasing number of applications of MaxSAT. To provide the reader with a better sense of the practical usefulness of MaxSAT, we first review some of these applications as well as some of the encoding techniques often applied to translate optimization problems into MaxSAT.

24.3.1. Encodings

Boolean optimization problems that can be expressed with linear constraints and a single optimization function are well-suited to be encoded and solved by MaxSAT solvers. However, it can be the case where the problem requires solving a multi-objective optimization function, or that the problem requires soft constraints to be modeled as a group of soft clauses instead of individual soft clauses. In those cases, MaxSAT can still be used with the encodings described in Sections 24.3.1.1 and 24.3.1.2, respectively. Moreover, problems that are originally encoded in different declarative optimization languages and approaches, such as MaxCSP, MinSAT, and pseudo-Boolean constraints can also be translated into MaxSAT, as overviewed in Section 24.3.1.3.

24.3.1.1. Boolean Multilevel Optimization

The class of Boolean optimization problems that can be represented using multi-objective functions with a predefined hierarchy of importance over the objective functions is known as *Boolean multilevel optimization (BMO)* [MAGL11].

Definition 3 (Boolean multilevel optimization). Boolean multilevel optimization (BMO) problems [MAGL11] consist of a set of sets of soft clauses $C = \{C_1, \dots, C_m\}$, where $\{C_1, \dots, C_m\}$ forms a partition of C , and each clause $c \in C_i$ has the same weight w_i . A MaxSAT formula is an instance of BMO if and only if the following condition holds:

$$w_i > \sum_{i+1 \leq j \leq m} w_j |C_j| \quad \text{with } i = 1, \dots, m-1.$$

In words, in BMO problems it is more valuable to satisfy a single soft clause in the set C_i than to falsify all soft clauses in the subsequent sets C_{i+1}, \dots, C_m .

Example 1. Consider the instance

$$F = \{(\neg x_1, x_2), (\neg x_1, x_4), (\neg x_1, \neg x_5), (\neg x_3, x_2, x_4), \\ (\neg x_1)_1, (\neg x_2)_1, (\neg x_3)_1, (\neg x_4)_1, (\neg x_5)_1\}.$$

Assume that instead of all soft clauses having the same priority, the user wants to give priority to satisfying some of the soft clauses over others. For example, say the user wants to solve a multilevel optimization problem where they have three objectives specified by $O_1 = \{(\neg x_1)\}$, $O_2 = \{(\neg x_2)\}$, and $O_3 = \{(\neg x_3), (\neg x_4), (\neg x_5)\}$,

with O_1 being strictly more important than O_2 and O_2 being strictly more important than O_3 . This problem can be solved with MaxSAT by associating suitable weights to each soft clause.

In this example, considering the formula F and the ordered optimization objectives O_1, O_2 and O_3 we would transform F into F^{BMO} by associating all soft clauses in O_3 with the weight 1. All soft clauses in O_2 with the weight equal to the sum of the weights of the soft clauses in O_3 plus 1 ($3 + 1 = 4$). Soft clauses in O_1 will have the weight 8 which corresponds to the sum of the weights of the soft clauses in O_2 and O_3 plus 1 ($3 + 4 + 1 = 8$). The intuition behind these weights is that the cost of falsifying a single soft clause in O_1 is greater than falsifying all soft clauses in O_2 and O_3 . Therefore, these costs capture the lexicographical order between the objective functions: $O_1 > O_2 > O_3$. We arrive at the instance

$$F^{\text{BMO}} = \{(\neg x_1, x_2), (\neg x_1, x_4), (\neg x_1, \neg x_5), (\neg x_3, x_2, x_4), \\ (\neg x_1)_8, (\neg x_2)_4, (\neg x_3)_1, (\neg x_4)_1, (\neg x_5)_1\}.$$

F^{BMO} can be solved using MaxSAT and the optimal solution to this problem corresponds to the optimal solution to the multilevel optimization problem described by the ordered objectives O_1, O_2 and O_3 .

24.3.1.2. Group MaxSAT

Some applications may require achieving more complex soft constraints that can only be represented with a set (or group) of clauses. In this case, the soft constraint is only satisfied if *all* of the clauses used to represent it are satisfied; i.e., the cost of the soft constraint will be incurred if any of its clauses are falsified. These kinds of soft clauses give rise to the *group MaxSAT* problem [HMM15]. In group MaxSAT there are k soft constraints where the i soft constraint is encoded by a group G_i of clauses. The goal in group MaxSAT is to find a solution that satisfies all hard clauses while maximizing the total weight of the satisfied soft constraints.

Group MaxSAT formulas can be transformed into an equivalent MaxSAT formulas by using specific encodings. One of these encodings is known as the \top -encoding [HMM15]. We refer the reader interested in other encodings of group MaxSAT to [HMM15].

Definition 4 (\top -encoding [HMM15]). A group MaxSAT instance F^{G} can be transformed into an equivalent MaxSAT instance F by adding all hard clauses from F^{G} to F . Let the i 'th soft constraint, with weight w_i , be represented by the group of soft clauses $G_i = c_1, \dots, c_k$. For each $c \in G_i$ we add the hard clause $(c \vee r_i)$ to F where r_i is a brand new variable. Finally, for each group G_i , we include the soft clause $(\neg r_i)_{w_i}$ to F .

Example 2. Consider the group MaxSAT instance

$$F^{\text{G}} = \{(\neg x_1, x_2), (\neg x_1, x_5), (\neg x_1, \neg x_4), (\neg x_3, x_2, x_4)\},$$

where

$$G_1 = (\{(\neg x_3), (\neg x_4), (\neg x_5)\})_2,$$

$$G_2 = (\{(\neg x_2), (\neg x_1)\})_3.$$

This formula can be transformed into an equivalent MaxSAT formula F by using the \top -encoding, resulting in

$$F = \{(\neg x_1, x_2), (\neg x_1, x_5), (\neg x_1, \neg x_4), (\neg x_3, x_2, x_4),$$

$$(\neg x_3, r_1), (\neg x_4, r_1), (\neg x_5, r_1), (\neg x_2, r_2), (\neg x_1, r_2)$$

$$(\neg r_1)_2, (\neg r_2)_3\}.$$

Note that the number of soft clauses of the transformed MaxSAT formula corresponds to the number of groups in the original group MaxSAT instance.

24.3.1.3. Encoding other Constraint Optimization Languages to MaxSAT

The algorithmic advances of MaxSAT make it appealing to encode problems from other domains into MaxSAT and solve them with MaxSAT solvers.

For example, other optimization problems such as the maximum constraint satisfaction problem (MaxCSP) [ACLM12] and minimum satisfiability (MinSAT) [ZLMA12, LZMS12] can be reduced to MaxSAT and solved with MaxSAT solvers.

Even decision problems such as SAT can be reduced to Horn MaxSAT by using the dual-rail encoding [IMM17]. This reduction coupled with MaxSAT resolution (Section 24.4.1.1.2) constitutes a proof system that has polynomial-time refutations for pigeonhole formulas. We refer the interested reader to [MIM17] for more examples of reductions of decision and optimization problems into Horn MaxSAT.

A common reduction is to transform a pseudo-Boolean optimization problem to MaxSAT. The interested reader is referred to Chapter 28 for more details on pseudo-Boolean optimization.

Definition 5 (Pseudo-Boolean optimization, PBO). Pseudo-Boolean optimization problems are of the following form:

$$\text{minimize } \sum_{j=1}^n c_j l_j$$

$$\text{subject to } \sum_{j=1}^n a_{ij} l_j \leq b_i \quad \forall i = 1, \dots, m,$$

where $n, m \in \mathbb{N}$ and $a_{ij}, b_i, c_j \in \mathbb{Z}$.

Example 3. Consider the PBO problem instance F^{PBO}

$$\text{minimize } x_1 + x_2 + x_3 + x_4 + x_5$$

$$\text{subject to } \quad \neg x_1 + x_2 \geq 1$$

$$\quad \quad \quad \neg x_1 + x_5 \geq 1$$

$$\quad \quad \quad \neg x_1 + \neg x_4 \geq 1$$

$$\quad \quad \quad \neg x_3 + x_2 + x_4 \geq 1$$

The formula F^{PBO} can be encoded to an equivalent MaxSAT formula F . The constraints are encoded as hard clauses by using appropriate cardinality and pseudo-Boolean encodings 24.4.2.1. The objective function can be encoded as soft clauses: for each literal $c_j x_j$ in the objective function, create a corresponding unit soft clause $(\neg x_j)_{c_j}$.

F^{PBO} is equivalent to the following MaxSAT formula

$$F = \{(\neg x_1, x_2), (\neg x_1, x_5), (\neg x_1, \neg x_4), (\neg x_3, x_2, x_4), \\ (\neg x_3)_1, (\neg x_4)_1, (\neg x_5)_1, (\neg x_2)_1, (\neg x_1)_1\}.$$

Notice that this is the same formula as the one presented in Example 1.

Note, however, that this example has a particularly simple translation to MaxSAT since all of the linear inequalities can be directly represented as clauses. In general, a linear constraint of the form $a_1 x_1 + a_2 x_2 + \dots + a_k x_k \geq t$ would have to be translated to CNF using a pseudo-Boolean constraint encoding (Section 24.4.2.1) which would add multiple clauses to the resultant MaxSAT formula.

24.3.2. Applications

With significant performance improvements in MaxSAT algorithms over the years there has been a corresponding increase in the number and diversity of applications of MaxSAT. Some examples of domains where MaxSAT has been successfully used include planning, scheduling and configuration problems, AI and data analysis problems, combinatorial problems, verification and security, and bioinformatics. We will briefly outline some of these applications and provide references to further details.

24.3.2.1. Planning, Scheduling, and Configuration

Planning problems [ZB12, MBM16] consist of finding a sequence of actions that transform an initial to a final state. These problems can be encoded to MaxSAT to efficiently find solutions that minimize the cost of the actions required to achieve the final state. MaxSAT has also been used to synthesize linear temporal logic (LTL) specifications for robot motion planning and control of autonomous systems in scenarios where specifications may be unrealizable and these can be split into soft and hard specifications [DGT18].

Scheduling problems are widespread and arise in various contexts. MaxSAT has been applied to solving different types of scheduling problems, including course timetabling [DM17, MNRS17, AN14], staff scheduling [DMW17, BGSV15, CHB17], and even wedding seating arrangements [MS17]. The course timetabling problem consists of creating a schedule such that it satisfies various constraints with respect to teachers, rooms, times, lectures and students. These scheduling problems are often solved manually by a school administrator but can be encoded into MaxSAT to lever the algorithmic advances for MaxSAT.

MaxSAT has also been used to solve configuration problems such as finding a pre-production vehicle configuration [MKTR16], solving the package upgradeability problem where the goal is to find new packages to be installed in the current system according to some user preference [ALM09, ABL⁺10b, IJM14], and even for seating arrangements at weddings [MS17].

24.3.2.2. AI and Data Analysis Problems

MaxSAT has been used in a variety of data-oriented AI problems, starting with computing most probable explanations (MPE) [Par02] in Bayesian networks, where the goal is to compute the most likely state of a Bayesian network given observations on a subset of the random variables. MaxSAT has been shown to yield competitive approaches to learning different classes of probabilistic graphical models, including Bayesian network structures with restricted treewidth [BJM14], causal discovery in very generic model spaces [HSJ17], as well as causal structure estimation from time series data [HPJ⁺17]. MaxSAT has also been proposed as an alternative approach to inferring problem-specific cutting planes in a state-of-the-art integer programming approach to Bayesian network structure learning [SMJ15]. Further recent data-oriented applications of MaxSAT include learning explainable decision sets [IPNM18] and interpretable classification rules [MM18], constrained correlation clustering [BJ13, BJ17], and neighborhood-preserving low-dimensional visualization of high-dimensional datasets [BJB⁺14].

Other AI applications, specifically within the area of knowledge representation and reasoning, consist of applying MaxSAT to understand the dynamics of argumentation frameworks [WNJ17, NWJ16a, NWJ16b] and in model-based diagnosis [MJIM15]. Given a model of a system and an input-output observation that is not consistent with the expected behavior, the goal of model-based diagnosis is to identify a subset of components that when removed make the system consistent with the observation. MaxSAT can be used to find the smallest set of components that need to be removed [MJIM15]. In the realm of abstract argumentation, MaxSAT enables efficiently reasoning about different notions of *enforcement*, dealing with smallest changes required to argumentation frameworks in light of knowledge on extensions [WNJ17, NWJ16a] as well as *synthesis* of argumentation frameworks based on negative and positive examples of extensions [NWJ16b].

24.3.2.3. Combinatorial Problems

MaxSAT can also be used to encode and solve many combinatorial problems, such as the Max-Clique problem [LQ10, FLQ⁺14, LJX15] where given a group of vertices, the maximal clique is the largest subset of vertices in which each point is directly connected to every other vertex in the subset.

Other combinatorial problems that have been encoded into MaxSAT include the Steiner tree problem [dOS15], tree-width computation [BJ14] and finding solutions for the maximum quartet consistency problem [MM10].

24.3.2.4. Verification and Security

Another domain with many successful MaxSAT applications is verification and security of software and hardware. For instance, MaxSAT can be used to design and debug circuits [SMV⁺07, CSVMS09, CSMV10, AIMT13, XRS03]. Other examples of applications of MaxSAT to hardware consist in solving the hardware-software partitioning problem which decides during the design phase which parts should be implemented either in hardware or software [TDFDGDJSJ⁺17].

MaxSAT also has many applications in security. For instance, solving the user authorization query problem [WQL09], reconstructing AES key schedule images [LZK16], detecting hardware Trojans [SA18], and maximizing the development assurance level to improve the rigor of the development of software and hardware on aircraft [BDS11]. Another application in security is to find malware in Android applications by maximizing the similarity between a malicious application and the application that is being scanned [FBM⁺17].

MaxSAT has also been used to minimize cascading style sheets (CSS) in order to reduce the size of the code transmitted over the web [HLH19]. Another application is to use MaxSAT for quality of service (QoS) of cloud services where the goal is to minimize the composition of distributed web services while satisfying the user preferences [WJH18, BACF17].

When analyzing software, program analysis tools usually trade precision for scalability and their output often contains undesirable output due to approximations. MaxSAT can be used in a user-guided program analysis setting where the hard clauses capture the soundness of the analysis and soft clauses capture the degrees of approximation and user's preferences [MZNN15a, SZGN17]. Encoding user-guided program analysis into MaxSAT can lead to very large formulas which can be solved by specialized MaxSAT solving algorithms [MZNN15b, ZMNN16].

MaxSAT can also be used for fault localization where the goal is to pinpoint the localization of software bugs [ZWM11, JM11]. In particular, MaxSAT can be used to reduce the error log reported to the programmer and decrease the amount of code that needs to be reviewed. The problem of state-space reduction for non-deterministic visibly pushdown automata can also be encoded into MaxSAT, which can be used to improve the performance of software model-checkers [HST17].

24.3.2.5. Bioinformatics

MaxSAT has applications in many interdisciplinary applications such as bioinformatics. One of these applications is to solve the haplotype inference problem which aims at finding the minimum number of haplotypes which explains a given set of genotypes [GML11, GMLO11].

Other applications within this domain consist of finding generalized Ising models [HKD⁺16], finding the maximum similarity between RNA sequences [Mar17], modeling biological networks representing the interactions between genes [GL12], and even applications in cancer therapy to find faulty areas of the gene regulatory network [LK12].

24.3.2.6. Further Applications

Yet another application of MaxSAT is to improve other constraint solving algorithms. For instance, MaxSAT can be used to restore satisfiability of constraint satisfaction problems (CSPs). By encoding a CSP instance as a MaxSAT instance, one can identify the smallest set of tuples to be removed from the CSP instance to restore satisfiability [LM11]. MaxSAT has also been used to enumerate minimal correction sets (MCS) [MLM13] with the goal of removing a minimal number of clauses such that an unsatisfiable SAT formula becomes satisfiable.

Propagation complete encodings have the property of being able to identify inconsistencies via unit propagation, thus allowing the encoding to be more efficiently solved by SAT solvers. However, this trade-off usually comes at the expense of larger SAT encodings. MaxSAT can be used to compute approximately propagation complete encodings [ER18] which allows conflicts to be detected earlier when using a SAT solver.

24.4. Modern MaxSAT Algorithms

In this section, we provide an overview of the currently most important algorithmic approaches to MaxSAT, focusing on SAT-based approaches: the model-improving approach (based on querying a SAT solver for solutions of increasing quality), the core-guided approach (in its several variants, based on extracting MaxSAT cores and compiling each core into the instance until a satisfying assignment is found), and the implicit hitting set approach (based on accumulating a set of MaxSAT cores and computing a minimum-cost hitting set of the cores until a satisfying assignment for the clauses not in the hitting set is found).

We will not cover branch-and-bound approaches to MaxSAT (e.g., [HLO08, LMMP10]). The interested reader is referred to Chapter 23 for an overview of such approaches. The main feature of the approaches we will present here is that they scale up to much larger problems than branch-and-bound approaches. There do exist small MaxSAT instances with a few hundred variables (e.g., finding largest cuts in random graphs) that branch-and-bound algorithms can solve very quickly (less than ten seconds) while the modern SAT-based MaxSAT algorithms overviewed in this chapter have considerable problem solving (i.e., instances remain unsolved after thousands of seconds). However, branch-and-bound approaches are typically ineffective on instances with more than a thousand variables. Problems from various application areas, including ones described in Section 24.3.2, are usually much larger, typically involving at least ten thousand variables and sometimes as many as millions of variables and clauses.

24.4.1. Background

While details on the MaxSAT formalism were already presented in Section 24.2, in order to properly explain current algorithms for solving MaxSAT we need some further background.

24.4.1.1. Transformations

MaxSAT algorithms rely on certain transformations that can be applied to a MaxSAT formula. There are a number of sound ways to transform a MaxSAT formula F into a new MaxSAT formula F' such that from an optimal solution to F' we can easily obtain an optimal solution for F . Such transformations include simplifying transformations used in preprocessing, and transformations used simply to make it more algorithmically convenient to solve the MaxSAT problem. Preprocessing is discussed in more detail in Section 24.5.1. In this section we present some more basic transformations that are useful for algorithmic convenience (both for MaxSAT solving and for further preprocessing).

24.4.1.1.1. The Blocking Variable Transformation. The blocking variable transformation is used on the input formula by a number of MaxSAT solvers. It transforms an input MaxSAT formula F into an equivalent formula F^b that has only unit soft clauses. Having a single literal ℓ whose truth value encodes the truth value of the soft clause c , allows us to use ℓ as an assumption to the SAT solver (Section 24.4.1.2) and to use ℓ as an input to cardinality constraints used when encoding the cost bound k .

In the F^b transformation, the hard clauses F are unchanged. Each soft clause c_i of F is transformed by adding the negation of a brand new variable b_i ,² adding the resultant extended clause $c_i \vee \neg b_i$ as a new hard clause, and creating a new unit soft clause (b_i) with weight equal to c_i 's weight (c_i 's weight is moved to the new unit soft clause).

Definition 6 (Blocking variable transformation). Given a MaxSAT instance F , F^b is a new MaxSAT instance with:

1. $h\ rd(F^b) = h\ rd(F) \cup \{(c_i \vee \neg b_i) \mid c_i \in o(F)\}$, where each b_i is a new variable called a **blocking** or a **relaxation** variable.
2. $o(F^b) = \{(b_i) \mid b_i \text{ was added in step 1}\}$
3. $w((b_i)) = w(c_i)$ (each new soft clause (b_i) gets the weight of the original $c_i \in o(F)$).

Example 4. If $F = \{(x, \neg y, z), (\neg y, z), (z, y)_{10}, (\neg z)_5\}$, then $F^b = \{(x, \neg y, z), (\neg y, z), (z, y, \neg b_1), (\neg z, \neg b_2), (b_1)_{10}, (b_2)_5\}$.³

F^b has the following useful properties.

1. F^b has only unit soft clauses, so there is a single literal denoting the truth and falsity of each soft clause.
2. If π^b is a feasible solution for F^b then $\pi^b|_{vars(F)}$ is a feasible solution for F with $o(\pi^b, F^b) \geq o(\pi^b|_{vars(F)}, F)$.
3. If π is a feasible solution for F , then we can extend π to π^b , a feasible solution for F^b , with $o(\pi, F) = o(\pi^b, F^b)$.

To illustrate property 2, $\pi^b = \{x, \neg y, z, \neg b_1, \neg b_2\}$ (listing the literals made *true* by π^b) is a feasible solution for F^b from Example 4, and $o(\pi^b, F^b) = 15$. Restricting π^b to $vars(F)$ we obtain $\pi^b|_{vars(F)} = \{x, \neg y, z\}$, which can be seen to be a feasible solution to F with lower cost $o(\pi^b|_{vars(F)}, F) = 5$.

To illustrate property 3, $\pi = \{x, \neg y, z\}$ is a feasible solution of F with $o(\pi, F) = 5$, and we can extend it to be a feasible solution of F^b by setting each b_i to *true* iff $\pi \models c_i$. In this case, we obtain $\pi^b = \{x, \neg y, z, b_1, \neg b_2\}$ with equal cost $o(\pi^b, F^b) = 5$.

Proposition 1. *Properties 2 and 3 hold for the blocking variable transformation.*

²The idea of adding a new variable to a clause so that the new variable encodes the clause's truth value appears in [GW93], but may have been used even earlier.

³For some MaxSAT algorithms it is possible in certain cases to avoid adding a new variable to soft clauses that are already unit [BSJ15b]. This can significantly reduce the number of new variables in the formula F^b making it more efficient to solve.

Proof. To see that property 2 is true, observe the following. If π^b is a feasible solution of F^b it must satisfy all hard clauses of F^b . Hence, π^b satisfies every hard clause of F since $h\text{ rd}(F) \subseteq h\text{ rd}(F^b)$. Since none of the hard clauses of F contain any new b_i variables we also have that $\pi^b|_{\text{vars}(F)}$ satisfies every hard clause of F , and thus $\pi^b|_{\text{vars}(F)}$ is a feasible solution for F . Consider $o(\pi^b|_{\text{vars}(F)}, F)$. This is the sum of the weights of all soft clauses falsified by $\pi^b|_{\text{vars}(F)}$. Let c_i be one of the soft clauses falsified by $\pi^b|_{\text{vars}(F)}$ so that weight $w(c_i)$ is a term in the sum $o(\pi^b|_{\text{vars}(F)}, F)$. Since π^b satisfies the hard clause $(c_i \vee \neg b_i)$ of F^b and $\pi^b|_{\text{vars}(F)}$ falsifies c_i , we must have that $\pi^b \models \neg b_i$. In other words, π^b must falsify the soft clause $(b_i) \in F^b$, which contributes the term $w((b_i)) (= w(c_i))$ to $o(\pi^b, F^b)$. That is, we must have that $o(\pi^b, F^b) \geq o(\pi^b|_{\text{vars}(F)}, F)$.

To see that property 3 is true, let π be a feasible solution for F . We extend π to π^b by setting $\pi^b \models b_i$ if $\pi \models c_i$ and $\pi^b \models \neg b_i$ if $\pi \not\models c_i$. As a result π^b will satisfy all hard clauses of F^b . Also, $o(\pi^b, F^b)$ will include the term $w((b_i))$ if and only if $o(\pi, F)$ includes the term $w(c_i) (= w(b_i))$. Hence $o(\pi, F) = o(\pi^b, F^b)$ \square

Furthermore, properties 2 and 3 imply that the blocking variable transformation MaxSAT equivalence in the following sense.

Proposition 2 (MaxSAT equivalence of F^b). *If π^b is an optimal solution of F^b then $\pi^b|_{\text{vars}(F)}$ is an optimal solution of F .*

Proof. The proposition follows from properties 2 and 3 on the blocking variable transformation. In particular (2) tells us that the cost of an optimal solution to F^b is at least as large as the cost of an optimal solution to F ; and (3) tells us that that is at least as small. \square

This proposition tells us that we can solve a MaxSAT formula F by solving F^b and then simply discarding the truth assignments to the newly introduced variables b_i .

Historically, the idea of adding “relaxation” or “blocking” variables to the soft clauses has been used in all MaxSAT algorithms based on solving a sequence of SAT instances. In the presentation of these algorithms, typically the relaxation variables are introduced at various stages of the algorithm’s execution rather than initially. In practice, however, MaxSAT solvers generally introduce the relaxation variables at the beginning, so as to exploit SAT solving under assumptions to extract cores, or to impose bounds on how many soft clauses can be falsified via cardinality or pseudo-boolean constraints. In general, it is simpler and more uniform to explain these different algorithms as they run in practice—i.e., as they run on F^b rather than as they run on F .

Also typically the positive form of the relaxation variables is added to the clause. So the soft clause $(x, y)_{10}$ would become the hard clause (x, y, b_1) , rather than $(x, y, \neg b_1)$ as we have done here. By adding the negation of the relaxation variable we avoid a potentially confusing polarity flip. In particular, now when b_i is *true* c_i must be satisfied, rather than having $\neg b_i$ imply that c_i must be satisfied.

24.4.1.1.2. Soft Clause Cloning and MaxSAT Resolution. Another transformation that is sometimes used when solving a MaxSAT formula is to split a soft clause c_{m+n} into two soft clauses c_m and c_n . For example, we can replace the single soft clause $(x, \neg y)_{12}$ with the two soft clauses $(x, \neg y)_5$ and $(x, \neg y)_7$. In the literature, this process is called soft clause cloning. It is not difficult to see that the cost of any feasible solution π remains unchanged by soft clause cloning (and hence the optimal solutions are unchanged). This transformation can also be performed in reverse, we can combine the weights of all copies of a soft clause into a single copy. Soft clause cloning is used in many (but not all) MaxSAT algorithms to deal with weighted instances.

Lifting the classical resolution rule to MaxSAT gives rise to a more complex transformation [LH05, BLM07]. The Larrosa and Heras [LH05] transformation is as follows. Given two clauses of the form $(x, A)_{w_1}$ and $(\neg x, B)_{w_2}$ and letting $m = \min(w_1, w_2)$ we can apply a MaxSAT resolution step as follows.

1. Replace the two original clauses $(x, A)_{w_1}$ and $(\neg x, B)_{w_2}$ by their residues:
 $(x, A)_{w_1-m}$ and $(\neg x, B)_{w_2-m}$.
2. Add the resolvent $(A, B)_m$.
3. Add the two compensation clauses $(x, A, \neg B)_m$ and $(\neg x, B, \neg A)_m$.

There are a few things to note. First, any soft clause with weight zero can be removed from the formula—such clauses cannot affect the cost of any feasible solution. Hence, at least one of the residue clauses will vanish. Second, the compensation clauses are not clauses (they contain conjunctions of literals $\neg A$ or $\neg B$). Hence, they have to be converted into a set of clauses, where the size of that set will depend on whether or not new variables are introduced. And third, by treating hard clauses as being soft clauses with infinite weight (e.g., $(x, y) = (x, y)_\infty$), we can apply MaxSAT resolution to combinations of soft and hard clauses.

Since the MaxSAT resolution rule of Larrosa and Heras can generate non-clausal formula it cannot form a proof system. In any proof system for CNF each rule of inference must yield a valid new CNF. Bonet et al. [BLM07] addressed this issue and gave a version of MaxSAT resolution that does yield a new CNF. They also showed that their version of MaxSAT resolution forms a complete proof system: for any MaxSAT instance F there exists a MaxSAT resolution proof that F has optimal cost $o(F)$. MaxSAT resolution has been used as a theoretical underpinning for some MaxSAT algorithms, e.g., PMRes [NB14] and MiniMaxSat [HLO08], but directly implementing MaxSAT resolution to solve MaxSAT does not seem to be practical [MM11].

24.4.1.2. Incremental and Assumption-based SAT Solving for Extracting Cores

An essential technique used in MaxSAT solving is assumption-based SAT solving [ES03] implemented in most modern SAT solvers. When given an input formula F a SAT solver can produce either a satisfying truth assignment or conclude that F is unsatisfiable. Assumption-based SAT solving extends the capacity of the SAT solver by asking it to solve F subject to a set of assumptions A which must be a set of literals. Now the SAT solver must either find a truth assignment satisfying $F \wedge \bigwedge_{\ell \in A} \ell$ (i.e., a truth assignment satisfying F that also makes all of the literals

in A true), or it must conclude that $F \wedge A$ is unsatisfiable. Furthermore, and most critical to MaxSAT solvers, when $F \wedge A$ are unsatisfiable the SAT solver is able to return a clause c such that

1. c contains only negated literals of A , i.e., $A \models \neg c$ (c is a conflict clause over A), and
2. $F \models c$.

The conflict clause c is thus a subset of A that is sufficient to show that no model of F satisfies all of A . In particular any model of F must falsify at least one of the literals of A whose negation is contained in c .

It should be noted that the conflict clause c returned by the SAT solver need not be minimal. That is, there could be another clause $c' \subsetneq c$ satisfying the above two conditions. Hence, if F by itself is unsatisfiable the SAT solver might return any subset of negated literals of A (including possibly the empty clause).

Assumptions can be used to extract cores when the MaxSAT formula contains unit soft clauses (e.g., after the F^b transformation has been applied). In particular, if $(b_1)_{w_1}, \dots, (b_m)_{w_m}$ are the soft clauses of a MaxSAT formula F then using $\{b_1, \dots, b_m\}$ as assumptions in a SAT solve of $h\ rd(F)$ will either result in a model of $h\ rd(F)$ that satisfies all of the soft clauses of F , i.e., a zero-cost model, or it will result in a conflict clause c such that $F \models c$ and $c \subseteq \{\neg b_1, \dots, \neg b_m\}$. That is, the conflict clause c asserts that at least one of the negated b_i literals in it must be true in any feasible model. In other words, c specifies a set of soft clauses at least one of which must be falsified by any feasible model, i.e., a core of F .

Assumptions also play a useful role in *incremental* SAT solving. In many approaches to MaxSAT solving a SAT solver is called on a sequence of problems that are closely related to each other. Each of problems could be solved by invoking a new instance of the SAT solver. However, then information computed during one SAT solver invocation (e.g., learnt clauses) cannot easily be exploited in subsequent solver invocations. The idea of incremental SAT solving is to use only one instance of the SAT solver for all of the problems so that all information computed can be retained for the next problem. In this case, we can monotonically add clauses to the SAT solver or use different sets of assumptions to specify each new problem to be solved. With assumptions we can add or remove certain clauses by adding to those clauses a new literal ℓ . When we assume $\neg \ell$ these clauses become active (added to the problem), and when we assume ℓ these clauses become inactive (removed from the problem).

24.4.2. Approaches based on Sequences of Satisfiability Queries

MaxSAT with weighted soft clauses is known to be in the complexity class FP^{NP} [Pap94]. This means that it can be solved with a polynomial number of calls to an NP-oracle. In particular, a polynomial number of calls to a SAT solver can be used to solve the MaxSAT instance F , with each SAT call determining if F can be satisfied by a feasible solution with cost less than or equal to k , for various values of k . With integer weights, if F can be satisfied with cost k and F cannot be satisfied with cost $k - 1$, then the feasible model achieving cost k must be an optimal solution.

Given the effectiveness of modern SAT solvers, the approach of solving MaxSAT via a sequence of calls to a SAT solver has proved to be both popular and effective. A number of different techniques have been developed for effectively encoding sequences of the following decision problem $\mathcal{D}(F, k)$, for various values of k , into CNF.

Definition 7 (The decision problem $\mathcal{D}(F, k)$). Let $\mathcal{D}(F, k)$ be the decision problem “does a feasible solution of F with cost less than or equal to k exist?”

In particular, we want to encode $\mathcal{D}(F, k)$, into a CNF formula that can be effectively solved by a SAT solver. The key to many of these encoding techniques is the successful exploitation of information returned by the SAT solver when it either determines $\mathcal{D}(F, k)$ to be unsatisfiable or satisfiable. In particular, when $\mathcal{D}(F, k)$ is satisfiable the cost of the model found can be used to determine the next value of k to test; and when the problem is unsatisfiable the returned **core** can be used to add additional constraints to the SAT solver so as to effectively encode $\mathcal{D}(F, k')$ for some value k' larger than k .

MaxSAT solvers using a sequence of SAT queries have been designed in a number of different ways. In this section, we will survey the main MaxSAT algorithms using this approach. First, however we introduce some important features of the CNF encoding of the decision problem “does a feasible solution of F with cost less than or equal to k exist”, since these features affect both the design and effectiveness of the MaxSAT algorithms discussed in this section.

24.4.2.1. Cardinality and Pseudo-Boolean Constraints

Pseudo-Boolean constraints are used to help encode $\mathcal{D}(F, k)$. A pseudo-Boolean constraint over the literals $\{\ell_1, \dots, \ell_m\}$ has the form $a_i \ell_1 + \dots + a_m \ell_m \leq k$ where the a_i and k are integers and the sum is evaluated by regarding $\ell_i = \text{true}$ as 1 and $\ell_i = \text{false}$ as 0.⁴ When all of the a_i are 1, the pseudo-Boolean constraint is called a cardinality constraint. Cardinality constraints limit the number of literals ℓ_i that can be *true*.

There is an extensive literature on different methods for encoding pseudo-Boolean constraints and cardinality constraints, e.g. [War98, FG10, ES06, BB03, Sin05, ANOR09, OLH⁺13, BBR09, MPS14, HMS12, JMM15]. The details of these encodings and their different properties will not be discussed here, but we will point out some key features of these encodings that are particularly relevant to their use in MaxSAT solvers.

The first feature of these encodings is their *size*, i.e., the number of clauses and new variables needed in the CNF encoding. The main parameters influencing size are m the number of literals whose sum is being taken, k the size of the upper bound, and $\max(a_i)$ the greatest coefficient in the sum.

For cardinality constraints the commonly used CNF encodings (e.g., the totalizer encoding [BB03]) the size of the encoding grows as $O(mk)$. So in the worst case when $k = m$ this can be as large as $O(m^2)$. Smaller CNF encodings for cardinality constraints exist. For instance, the modulo totalizer encoding [OLH⁺13]

⁴The other relations ($=$, \geq , $<$, and $>$) can also be represented.

requires $O(m^{\frac{3}{2}})$ clauses, whereas the cardinality network encoding [ANOR09] requires $O(m \log_2(k))$ clauses. However, there are many MaxSAT problems with over 100,000 soft clauses. After the F^b transformation this would yield over 100,000 soft unit clauses. So we can see that naively constructing a cardinality constraint summing all of these literal might require as much as 10^{10} clauses. So MaxSAT solvers must exploit cardinality constraints in more clever ways.

For pseudo-Boolean constraints, the typical encodings are much larger. Some encodings do not necessarily depend on k but rather on m and $\log_2(\max(a_i))$ (the number of bits required to represent the coefficients in binary). There are pseudo-Boolean encodings of size as small as $O(m \log_2(\max(a_i)))$ (linear in m if the size of the coefficients is bounded) [War98], and others that are of larger size $O(m^3 \log_2(m) \log_2(\max(a_i)))$ [BBR09], $O(m^2 \log_2(m) \log_2(\max(a_i)))$ [MPS14]. Other encodings may depend on k (e.g. the sequential weight counter encoding [HMS12]) and have size $O(mk)$, or depend on the number of distinct weights (which is the case for e.g. the generalized totalizer encoding [JMM15]). The tradeoff between these encodings is that the larger encodings achieve better propagation, as we discuss next.

The second relevant feature of these encoding is their propagation power. This is the extent to which unit propagation can find entailed literals. Cardinality constraint encodings typically allow unit propagation to achieve *arc consistency* [Dec03, BB03]. That is, unit propagation on the encoding is able to find all literals entailed by the cardinality constraint. Since SAT solvers learn clauses based on the set of unit propagated literals, this degree of propagation power enhances the effectiveness of the SAT solver. The watchdog encoding [BBR09] was the first polynomial size pseudo-Boolean encoding that maintains arc consistency. Since then, several other polynomial arc consistent encodings have been proposed [MPS14, HMS12]. Encodings that achieve arc consistent are typically very large. Nevertheless, if the encoding is not prohibitively large then it can have better SAT solving performance due to greater propagation power.

Finally, the third relevant feature is whether or not the encoding can be made incremental. That is, given a CNF encoding of $a_i \ell_1 + \dots + a_m \ell_m \leq k$ can we incrementally add additional clauses to encode $a_i \ell_1 + \dots + a_m \ell_m + a_{m+1} \ell_{m+1} \dots a_{m+j} \ell_{m+j} \leq k'$, where we are summing over more literals and $k' \geq k$. For cardinality constraints incrementality was achieved in [MML14], and for pseudo-Boolean constraints [MJML14a, PRB18] reported on incremental ways of changing the left-hand size k to k' . In both cases making these encodings incremental yielded significant speedups for the underlying MaxSAT solver.

A critical element supporting incremental SAT solving (Section 24.4.1.2) is that some cardinality constraint encodings (e.g., the totalizer encoding [BB03]) include a set of “output” variables that encode the sum of the input literals in unary. That is, the encoding has a set of “output” variables o_i such that for any i , o_i is *true* if and only if the input literals sum to at least i . This allows using incremental SAT solving when changes are made to the the left-hand side of the constraint simply by making different assumptions. For example, given the totalizer encoding of the cardinality constraint $\sum_{j=1}^m \ell_j \leq k$ we can incrementally impose the constraint $\sum_{j=1}^m \ell_j < i$ for any $i \leq k$, by simply assuming $\neg o_i$ during the SAT call.

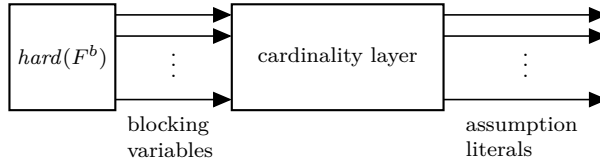


Figure 24.1: Structure of CNF for solving $\mathcal{D}(F, k)$. Note some blocking variable inputs to the cardinality layer might be passed directly through to the outputs (i.e., used directly as assumption literals).

24.4.3. General Structure of CNF Encodings of $\mathcal{D}(F, k)$

Figure 24.1 shows the general structure of the CNF formulas typically used for encoding the decision problem $\mathcal{D}(F, k)$. As already mentioned, the MaxSAT algorithms discussed in this section are all based on solving a sequence of $\mathcal{D}(F, k)$ decision problems for various values of k using a SAT solver. The encoding typically used consists of the hard clauses of the blocking variable transformation, $hard(F^b)$, conjoined with clauses of a cardinality layer. The clauses of the cardinality layer encode various cardinality constraints (or pseudo-Boolean Constraints) over the *blocking* variables of F^b , and sometimes over other internal variables of the cardinality layer.

The soft clauses of F^b are all of the form (b_i) , i.e., positive instances of the newly introduced blocking variables. Typically, the cardinality layer imposes constraints on which and how many of the literals $\neg b_i$ can be *true*. That is, it imposes constraints on the set of soft clauses of F^b that the SAT solver is allowed to falsify. In addition, various literals from the cardinality layer (including perhaps some of blocking variables) can be assumed to be *true* in assumption-based SAT solving.

The SAT solver, for which all clauses are of course hard, has all of the clauses of $hard(F^b)$. Hence, all satisfying models found by the SAT solver must be feasible models of F when restricted to $var(F)$. (Since these models are feasible solutions of F^b this follows from the properties of F^b described in Section 24.4.1.1.1). Furthermore, the SAT solver must find a model that satisfies all of the clauses of the cardinality layer and makes its current set of assumptions *true*. Of course, if no such model exists the SAT solver can return a conflict clause over its assumptions, as described in Section 24.4.1.2.

24.4.4. Model-Improving Algorithms

The first MaxSAT algorithm we present is the model-improving Linear SAT/UNSAT (LSU) algorithm. LSU follows almost directly from a proof that MaxSAT is in the complexity class FP^{NP} . That is, it uses a SAT solver to solve the sequence of decision problems $\mathcal{D}(F, k_0), \mathcal{D}(F, k_1), \dots, \mathcal{D}(F, k_u)$, where $k_0 = \infty$ (i.e., find a feasible solution irrespective of cost), k_i is the cost of the model found when solving $\mathcal{D}(F, k_{i-1})$ minus one, and $\mathcal{D}(F, k_u)$ is the first decision problem in the sequence that was unsatisfiable (UNSAT). The feasible solution returned by

Algorithm 1: Linear SAT/UNSAT algorithm for MaxSAT [BP10]

```

1 LSU( $F$ )
2  $F^b = \text{bv\_transform}(F)$ ;  $\text{card\_layer} = \{\}$ ;  $\text{Blits} = \{-b_i \mid (b_i) \in \text{soft}(F^b)\}$ 
3  $\text{bestmodel} = \{\}$ ;  $\text{cost} = \infty$ ;  $\text{sat?} = \text{true}$ 
4 while  $\text{sat?}$  do
    | /* SAT solver returns boolean SAT/UNSAT status */
    | /* if SAT returns  $\pi$ , the model found */
5    $(\text{sat?}, \pi) = \text{SATSolve}(\text{hard}(F^b) \cup \text{card\_layer})$ 
6   if  $\text{sat?}$  then
7     |  $\text{bestmodel} = \pi$ 
8     |  $\text{cost} = \text{cost}(\pi)$ 
9     |  $\text{card\_layer} = \text{CNF}(\sum_{b_i \in \text{Blits}} \text{wt}((b_i)) \times -b_i < \text{cost})$ 
10  else
11  | return  $(\text{bestmodel}, \text{cost})$ 

```

the last satisfiable instance $\mathcal{D}(F, k_{u-1})$ when restricted to the variables of F is an optimal solution of F .

Algorithm 1 gives the details. First any feasible model π is found (line 5) with card_layer equal to the empty set of clauses. Then using either a cardinality (when $w((b_i)) = 1$ for all i) or pseudo-Boolean constraint (when the soft clauses have varying weights) in the cardinality layer (line 9), the SAT solver is asked to find a feasible solution with cost strictly less than $\text{cost}(\pi)$.⁵ This is repeated until the SAT solver returns UNSAT, in which case the last feasible solution returned is an optimal solution.

The approach of solving a sequence of decision problems $\mathcal{D}(F, k_i)$ with decreasing bounds k_i has a long history. The earliest reported use of this approach in SAT solving appears to be [ARMS02]. The approach was also used in the Minisat+ solver described in [ES06], and in the Sat4j MaxSAT solver [BP10] where the name LSU was first used and an algorithm similar to Algorithm 1 was presented. These three systems actually found optimal models with respect to pseudo-Boolean objective functions. However, pseudo-Boolean objective functions can easily represent MaxSAT when all of the soft clauses are unit (as they are in the transformed formula F^b).

This approach continues to be used in the QMaxSAT [KZFH12] and Pacose solvers (described in [BJM18]). QMaxSAT includes a number of different cardinality and pseudo-Boolean encodings, and carefully chooses which encoding to use based on the characteristics of the input problem. Pacose builds on QMaxSAT by implementing a dynamic construction of one of its pseudo-Boolean encodings [PRB18].

Recently, the LSU algorithm has seen increasing use in incomplete MaxSAT solving in which the aim is to return the lowest-cost feasible solution within a given time bound. As can be seen from Algorithm 1 its chief benefit in this context is that each completed call to the SAT solver returns an improved solution. Hence, at any point the best model found so far can be returned.

Overall, however, the downside of this algorithm is that the cardinality or

⁵Finding a low cost initial feasible model can potentially reduce the size of the cardinality constraint encoding (see Section 24.4.2.1).

pseudo-Boolean constraint needs to be over m literals, where m is the number of soft clauses. As pointed out in Section 24.4.2.1, this can lead to very large encodings when the MaxSAT formula has tens or hundreds of thousands of soft clauses. Hence, solving the resulting SAT problem can become infeasible.

24.4.5. Core-Guided Algorithms

In contrast to LSU, core-guided algorithms work from UNSAT to SAT. That is, they solve a sequence of decision problems $\mathcal{D}(F, k_i)$ where k_i is increasing ($k_i > k_{i-1}$). When k_i is less than the cost of an optimal solution, $\mathcal{D}(F, k_i)$ will be unsatisfiable. Essentially, core-guided algorithms increase k_i until the current $\mathcal{D}(F, k_i)$ is satisfiable, but do not do this as explicitly as done in the LSU approach.

As with LSU these algorithms utilize information from the SAT solver returned from each UNSAT result, to construct the next decision problem to solve. The key insight, originally coming from Fu and Malik [FM06] is that cores can be extracted from the SAT solver when $\mathcal{D}(F, k)$ is unsatisfiable. Every feasible solution must falsify at least one soft clause of every core. Hence, the cores provide us information about the subset of soft clauses that the SAT solver can consider for falsification when solving $\mathcal{D}(F, k)$.

Empirically, many MaxSAT instances have optimal solutions falsifying very few soft clauses (relative to the total number of soft clauses). Hence MaxSAT cores often contain only a small subset of the soft clauses. So instead of having large cardinality constraints over all of the soft clauses, core-based algorithms can often utilize much smaller cardinality constraints over only those soft clauses appearing in the found cores, as these are the only ones that the SAT solver needs to consider falsifying. The other soft clauses can be required to be satisfied by using assumptions, rather than via cardinality constraints.

24.4.5.1. Fu-Malik

Fu and Malik presented the first MaxSAT algorithm exploiting cores [FM06]. Algorithm 2 gives the details. This algorithm only works on unweighted MaxSAT instances (i.e., all soft clauses have weight 1).

All blocking variables are always assumed to be *true* (line 2 and 5). Their main role is simply to allow the SAT solver to extract cores via its assumption mechanism. In the initial call, `card_layer` is empty. Hence, the initial call is asking the SAT solver to find a feasible model satisfying all soft clauses (line 5). In particular, every soft clause c_i of F has been encoded in F^b as the hard clause $(c_i \vee \neg b_i)$. Hence, when we assume b_i this clause is reduced to c_i , and the SAT solver must satisfy it.

Unless there is a zero-cost feasible solution, this call will result in the SAT solver declaring unsatisfiability. Furthermore, the SAT solver will also identify a subset of negated assumptions forming a core, κ . That is, at least one of the literals $\neg b_i$ in κ must be *true*. The algorithm then modifies the hard clause corresponding to each of the soft clauses in the core; for $\neg b_i$ this is the hard clause $(c_i \vee \neg b_i)$. It adds a brand new relaxation variable r_i^{cost} to each such clause by modifying the c_i part of the clause (line 8). (Note that “cost” is always incremented in the loop, line 10, so different variables are added in each iteration).

Algorithm 2: Fu-Malik algorithm for unweighted MaxSAT [FM06]

```

1 FuMalik( $F$ )
2  $F^b = \text{bv\_transform}(F)$ ;  $\text{card\_layer} = \{\}$ ;  $\text{assumptions} = \{b_i \mid (b_i) \in \text{soft}(F^b)\}$ 
3  $F' \text{ hard}(F^b)$ ;  $\text{cost} = \text{ ; sat?} = \text{false}$ 
4 while not sat? do
    /* SAT solver takes as input the CNF to solve and a set of literals to assume */
    /* Returns boolean SAT/UNSAT status */
    /* If SAT returns  $\pi$ , the model found */
    /* If UNSAT returns  $\kappa$ , a core (subset of negated assumptions). */
5 ( $\text{sat?}, \pi, \kappa$ ) = SATSolve ( $F' \cup \text{card\_layer}$ ,  $\text{assumptions}$ )
6 if not sat? then
7     for  $\neg b_i \in \kappa$  do
8          $c_i \quad c_i \vee r_i^{\text{cost}}$ 
            /* Modifying  $c_i$  modifies the clause  $(c_i \vee b_i) \in F'$  */
            /* Each  $r_i^{\text{cost}}$  is a brand new variable. */
9          $\text{card\_layer} = \text{card\_layer} \cup \text{CN} \left( \sum_{\{i \mid \neg b_i \in \kappa\}} r_i^{\text{cost}} \right)$ 
10         $\text{cost} = \text{cost} + 1$ 
11 else
12 return ( $\pi|_{\text{vars}(F)}$ ,  $\text{cost}$ )

```

Finally, it adds a cardinality constraint asserting that one and only one of the newly added relaxation variables can be made *true*. Making r_i^{cost} *true* immediately satisfies (or relaxes) the clause $(c_i \vee r_i^{\text{cost}} \vee \neg b_i)$, allowing the SAT solver to set the assumption b_i to *true* with impunity.

This algorithm does not quite follow the general structure described in Section 24.4.3. In particular, each new call to the SAT solver involves changing $\text{hard}(F^b)$. Since the change involves adding brand new variables to existing clauses (line 8), it is not easy to use the SAT solver incrementally in this algorithm.

Example 5. Consider $F = \{(x, \neg y), (y, z), (y, \neg z), (\neg x)_1, (\neg y)_1\}$. In this formula both x and y are forced, and thus both soft clauses must be falsified. Any feasible solution has cost 2, and thus all feasible solutions are optimal solutions.

$F^b = \{(x, \neg y), (y, z), (y, \neg z), (\neg x, \neg b_1), (\neg y, \neg b_2), (b_1)_1, (b_2)_1\}$. The first iteration of the main loop of Algorithm 2 would call the SAT solver on the CNF $\{(x, \neg y), (y, z), (y, \neg z), (\neg x, \neg b_1), (\neg y, \neg b_2)\}$ with the assumptions $\{b_1, b_2\}$. A possible core is $\kappa = \{\neg b_1, \neg b_2\}$. (Note that the cores returned by the SAT solver do not have to be minimal.)

The second iteration would then be a call to the SAT solver on the CNF $\{(x, \neg y), (y, z), (y, \neg z), (\neg x, \mathbf{r}_1^0, \neg b_1), (\neg y, \mathbf{r}_2^0, \neg b_2)\} \cup \text{CNF}(r_1^0 + r_2^0 = 1)$ with the same assumptions $\{b_1, b_2\}$. Again $\kappa = \{\neg b_1, \neg b_2\}$ is a possible core.

The third iteration would then be a call to the SAT solver on the CNF $\{(x, \neg y), (y, z), (y, \neg z), (\neg x, \mathbf{r}_1^0, \mathbf{r}_1^1, \neg b_1), (\neg y, \mathbf{r}_2^0, \mathbf{r}_2^1, \neg b_2)\} \cup \text{CNF}(r_1^0 + r_2^0 = 1) \cup \text{CNF}(r_1^1 + r_2^1 = 1)$ with the same assumptions $\{b_1, b_2\}$. This time the formula is satisfiable. For example, the truth assignment $\{x, y, z, r_1^0, r_2^1, \neg r_2^0, \neg r_1^1, b_1, b_2\}$ satisfies all of the clauses. At this point $\text{cost} = 2$.

Each core yields a subset of soft clauses of which at least one must be falsified. Adding the relaxation variables and the cardinality constraint gives the SAT

Algorithm 3: WPM1/WMSU1 algorithm for MaxSAT [ABL09, MMP09]

```

1 WPM1/WMSU1( $F$ )
2  $F^b = \text{bv\_transform}(F)$ ;  $\text{card\_layer} = \{\}$ ;  $\text{assumptions} = \{b_i \mid (b_i) \in \text{soft}(F^b)\}$ 
3  $F' = \text{hard}(F^b)$ ;  $\text{cost} = 0$ ;  $\text{iter} = 0$ ;  $\text{sat?} = \text{false}$ 
4  $m = \max\{i \mid b_i \in \text{assumptions}\} + 1$ 
5 while not sat? do
6   /* SAT solver interface is the same as in Algorithm 2 */
7    $(\text{sat?}, \pi, \kappa) = \text{SATSolve}(F' \cup \text{card\_layer}, \text{assumptions})$ 
8   if not sat? then
9      $wt_{\min} = \min_{\{b_i \mid \neg b_i \in \kappa\}} wt((b_i))$ 
10     $\text{cost} = \text{cost} + wt_{\min}$ 
11    for  $\neg b_i \in \kappa$  do
12      if  $wt((b_i)) > wt_{\min}$  then
13        /* Add new copy of clause  $c_i$  with new blocking variable  $b_m$  */
14         $c_m = c_i$ ;  $m = m + 1$ ;  $F' = F' \cup \{(c_m \vee \neg b_m)\}$ 
15         $\text{assumptions} = \text{assumptions} \cup \{b_m\}$ 
16         $wt((b_m)) = wt((b_i)) - wt_{\min}$  /* New copy gets residual weight */
17         $wt((b_i)) = wt_{\min}$  /* Old copy gets weight  $wt_{\min}$  */
18         $c_i = c_i \vee r_i^{\text{iter}}$  /* Add new relaxation variable to old copy */
19        /* Modifies clause  $(c_i \vee \neg b_i) \in F'$  */
20     $\text{card\_layer} = \text{card\_layer} \cup \text{CN}(\sum_{\{i \mid \neg b_i \in \kappa\}} r_i^{\text{iter}})$ 
21     $\text{iter} = \text{iter} + 1$ 
22  else
23    return  $(\pi|_{\text{vars}(F)}, \text{cost})$ 

```

solver the freedom to falsify one (and only one) of these soft clauses by setting its relaxation variable to *true*. Each iteration finds a core subject to all of the previously derived relaxations. That is, at least one clause from the i 'th core κ_i must be falsified even when we falsify a soft clause from each previously derived core. Hence, at stage k we have proved that the optimal cost is at least k . Note that the cores allow us to restrict the set of soft clauses that need to be considered for falsification. That is, only those soft clauses appearing in the found cores need be considered for falsification. This makes the size of the cardinality constraint encodings much smaller.

The example shows that one feature of this algorithm is that it can add multiple relaxation variables to the same clause if that clause appears in more than one core. This is a by-product of using an “= 1” cardinality constraint. In the example, we need to use one relaxation variable from each cardinality constraint in order to satisfy the two clauses $(\neg x, \neg b_1, \mathbf{r}_1^0, \mathbf{r}_1^1)$ and $(\neg y, \neg b_2, \mathbf{r}_2^0, \mathbf{r}_2^1)$; we cannot use two relaxation variables from the same cardinality constraint. These multiple relaxation variables in the soft clause can be a source of significant inefficiency as they introduce symmetries that can slow the SAT solver down [ABL13].

24.4.5.2. WPM1/WMSU1—Fu-Malik for Weighted MaxSAT

A restriction of the Fu-Malik algorithm is that it only works with unweighted MaxSAT instances. Soft clause cloning is a technique [ABL09, MMP09] that allows many unweighted algorithms to be extended to the general weighted case.

This technique was first used in the WPM1 [ABL09] and WMSU1 [MMP09] algorithms which appeared concurrently and independently in 2009. Both algorithms are essentially identical and extend Fu-Malik to deal with weighted soft clauses. Details are presented as Algorithm 3.

When a new core κ is found after we have already found and relaxed some previous cores, we know that at least one of the soft clauses in κ must be falsified. If these soft clauses have different weights, all that can be concluded is that the lower bound on the cost of an optimal solution must be incremented by $w_{\min} = \min_{\{-b_i \in \kappa\}} w((b_i))$, i.e., by the minimum weight among the soft clauses in κ . Clause cloning takes place at this stage, converting κ into a core containing clauses of weight w_{\min} by splitting every clause in κ into two copies: one with weight w_{\min} and one with the residual weight. The copies with the residual weight are added to the set of soft clauses, while the copies with weight w_{\min} are processed as an unweighted core.

In Algorithm 3 we again use the blocking variables introduced into F^b as assumptions so that we can extract cores. In processing each core we add a new relaxation variable to the hard clause corresponding to each of the soft clauses in the core (line 16). And for those soft clauses with weight greater than the minimum weight in the core (w_{\min}), we create a new copy of the clause (line 12), give it a brand new blocking variable b_m , and add it to the formula F' (line 12). The old soft clause (b_i) , whose hard clause $(c_i \vee r_i^{\text{iter}} \vee \neg b_i)$ contains the new relaxation variable, gets weight w_{\min} . The new soft clause (b_m) , whose hard clause $(c_m \vee \neg b_m)$ is a copy of the original clause with a new blocking variable, gets the remaining weight $w((b_i)) - w_{\min}$ (line 14). Note that, as in Algorithm 2, the unit soft clauses of F^b , including the new soft clause (b_m) , only appear in the set of assumptions. So we also need to add b_m to the set of assumptions (line 13). This means that the SAT solver is not allowed to falsify the original soft clause c_i until this clause has appeared in cores whose sum of w_{\min} weights is equal to $w(c_i)$. Only at that point will all copies of c_i have a relaxation variable allowing the SAT solver to falsify c_i . Furthermore, to falsify c_i the SAT solver will have to use a relaxation variable from each copy (by setting that variable to *true*). This will then “use up” all of the cardinality constraints c_i appeared in. That is, once c_i is *false*, all of the cardinality constraints it appears in will have one relaxation variable set, and will thus disallow any other relaxation variable in the constraint from being used.

Example 6. Consider

$$\begin{aligned} F &= \{(\neg x, \neg q), (\neg y, \neg q), (\neg z, \neg q), (x)_1, (y)_1, (z)_1, (q)_2\}, \\ F^b &= \{(\neg x, \neg q), (\neg y, \neg q), (\neg z, \neg q), (x, \neg b_1), (y, \neg b_2), (z, \neg b_3), (q, \neg b_4), \\ &\quad w((b_1)) = w((b_2)) = w((b_3)) = 1, w((b_4)) = 2\}. \end{aligned}$$

On this input, Algorithm 3 could perform the following sequence of iterations (different iterations are possible depending on what cores are returned by the SAT solver).

1. Core $\kappa_1 = \{-b_1, \neg b_4\}$, then $w_{\min} = 1$, $\text{cost} = 0 + w_{\min} = 1$, and

$$F' = \{(x, \neg q), (y, \neg q), (z, \neg q), (x, \mathbf{r}_1^0, \neg b_1), (y, \neg b_2), (z, \neg b_3), \\ \mathbf{r}_4^0, \neg b_4), (\mathbf{q}, \neg \mathbf{b}_5)\} \cup \mathbf{CNF}(\mathbf{r}_1^0 + \mathbf{r}_4^0 = 1), \\ \text{assumptions} = \{b_1, b_2, b_3, b_4, \mathbf{b}_5\}, \\ w((b_1)) = w((b_2)) = w((b_3)) = 1, \mathbf{wt}((\mathbf{b}_4)) = 1, \mathbf{wt}((\mathbf{b}_5)) = 1.$$

2. Core $\kappa_2 = \{-b_1, \neg b_2, \neg b_4, \neg b_5\}$, then $w_{\min} = 1$, $\text{cost} = 1 + w_{\min} = 2$, and

$$F' = \{(x, \neg q), (y, \neg q), (z, \neg q), (x, r_1^0, \mathbf{r}_1^1, \neg b_1), (y, \mathbf{r}_2^1, \neg b_2), (z, \neg b_3), \\ (q, r_4^0, \mathbf{r}_4^1, \neg b_4), (\mathbf{q}, \mathbf{r}_5^1, \neg \mathbf{b}_5)\} \\ \cup \mathbf{CNF}(r_1^0 + r_4^0 = 1) \cup \mathbf{CNF}(\mathbf{r}_1^1 + \mathbf{r}_2^1 + \mathbf{r}_4^1 + \mathbf{r}_5^1 = 1), \\ \text{assumptions and weights unchanged.}$$

3. Now the formula is satisfied by the truth assignment $\pi = \{x, y, z, \neg q, r_4^0, r_5^1, \neg r_1^0, \neg r_1^1, \neg r_2^1, \neg r_4^1, b_1, b_2, b_3, b_4, b_5\}$, and $\pi|_{\text{vars}(F)} = \{x, y, z, \neg q\}$ is an optimal solution for F .

Fu-Malik and its weighted version are mostly of historical interest as the first application of cores and clause cloning, respectively. To date no one has developed implementations of these algorithms that perform as efficiently as more recent algorithms. It is of course always possible that new insight could make these algorithms competitive.

24.4.5.3. MSU3 for Unweighted MaxSAT

MSU3 [MP07] is a much simpler algorithm than Fu-Malik (Section 24.4.5.1), and, when implemented with incremental SAT solving [MJML14b], it has proved to be much more efficient. In fact, at the time this chapter was written, the implementation of MSU3 in the Open-WBO system [MML14] was one of the most effective solvers for the unweighted MaxSAT problem. MSU3, like the LSU algorithm (Section 24.4.4) and unlike the Fu-Malik algorithm, uses a simple encoding that transparently captures the decision problem $\mathcal{D}(F, k)$. Like Fu-Malik, MSU3 starts with $k = 0$ and increases k until $\mathcal{D}(F, k)$ becomes SAT exploiting the cores found for each UNSAT decision problem instance $\mathcal{D}(F, k)$. In particular, in the unweighted case, $\mathcal{D}(F, k)$ is asking the SAT solver to find a feasible solution that falsifies at most k soft clauses. The SAT solver need not consider all soft clauses for falsification. Rather, it needs to consider only those soft clauses that have appeared in one of the cores obtained from the previous SAT solver calls. This makes the cardinality constraint used in the encoding of $\mathcal{D}(F, k)$ much smaller and easier for the SAT solver to handle.

Algorithm 4 gives the details. It starts with an empty `card_layer` and assumptions that force all soft clauses to be satisfied (line 2). Hence, the SAT solver is being asked to find a feasible solution with zero cost. Unless a solution is found, the SAT solver will return a core specifying a set of soft clauses one of which must be falsified. The literals b_i forcing the satisfaction of these soft clauses (b_i) are then removed from the assumptions (line 9), and added to the cardinality constraint (line 8). Finally, the cost is incremented and a new cardinality constraint is constructed specifying that at most this new “cost” of soft clauses can be falsified.

Algorithm 4: MSU3 algorithm for unweighted MaxSAT [MP07]

```

1 MSU3( $F$ )
2  $F^b = \text{bv\_transform}(F)$ ;  $\text{card\_layer} = \{\}$ ;  $\text{assumptions} = \{b_i \mid (b_i) \in \text{soft}(F^b)\}$ 
3  $\text{inCard} = \{\}$ 
4  $F' = \text{hard}(F^b)$ ;  $\text{cost} = 0$ ;  $\text{sat?} = \text{false}$ 
5 while not sat? do
6     /* SAT solver interface is the same as in Algorithm 2 */
7      $(\text{sat?}, \pi, \kappa) = \text{SATSolve}(F' \cup \text{card\_layer}, \text{assumptions})$ 
8     if not sat? then
9          $\text{inCard} = \text{inCard} \cup \kappa$ 
10         $\text{assumptions} = \text{assumptions} \setminus \{b_i \mid \neg b_i \in \kappa\}$ 
11         $\text{cost} = \text{cost} + 1$ 
12         $\text{card\_layer} = \text{CN}(\sum_{\neg b_i \in \text{inCard}} \neg b_i \leq \text{cost})$ 
13    else
14        return  $(\pi|_{\text{vars}(F)}, \text{cost})$ 

```

Note that the initial cardinality constraint is only over the soft clauses that have appeared in the found core κ . In subsequent iterations, the core κ will specify a subset of soft clauses one of which must be falsified *even though the cardinality constraint already allowed some number of soft clauses to be falsified*. Hence, the bound on the cardinality constraint is always incremented by one, and it is always over the soft clauses in the union of the cores that have been found so far. Note also that by removing the literal b_i from the assumptions, we no longer require the SAT solver to satisfy the soft clause (b_i) . However, by adding $\neg b_i$ to the cardinality constraint we still require the SAT solver to not falsify more than *cost* of these soft clauses. Note also that κ can be empty. In this case, the SAT solver has proved that the current bound on the number of soft clauses that can be falsified in the current cardinality constraint is too low. In this case, the only change to card_layer will be to increase the right-hand side bound “cost”.

If we consider the formula being solved by the SAT solver on line 6, we see that only card_layer changes between SAT solver invocations. Furthermore, card_layer changes in two ways: (a) the right-hand side cost bound is incremented, and (b) potentially additional literals are added to the left-hand side sum. Martins et al. [MJML14b] show how, using a totalizer encoding of the cardinality constraint [BB03], both changes can be made by only adding new clauses to the SAT solver and by changing the assumed totalizer output variable to change the right-hand side bound (Section 24.4.2.1). Hence, the same SAT solver instance can be used throughout the algorithm giving rise to significant performance improvements over non-incremental implementations.

24.4.5.3.1. The PM2 Algorithm. Algorithm 4 (MSU3) uses a single cardinality constraint over all of the soft clauses that have appeared in some core. However, in some cases, this cardinality constraint can be decomposed into smaller cardinality constraints each over a disjoint set of soft clauses. This is the insight used in the PM2 algorithm [ABL09, ABL13]. In particular, PM2 operates just like MSU3 except that in each iteration card_layer will contain a set of cardinality constraints rather than just one larger cardinality constraint.

Consider the following sequence of cores (discovered by the SAT solver in this order): $\kappa_1 = \{\neg b_1, \neg b_2, \neg b_3\}$, $\kappa_2 = \{\neg b_4, \neg b_5, \neg b_6\}$, $\kappa_3 = \{\neg b_7, \neg b_8\}$, $\kappa_4 = \{\neg b_1, \neg b_6\}$. MSU3 would construct the following sequence of cardinality constraints (setting `card_layer` to the i 'th constraint in the i 'th iteration):

- (1) $\neg b_1 + \neg b_2 + \neg b_3 \leq 1$,
- (2) $\neg b_1 + \dots + \neg b_6 \leq 2$,
- (3) $\neg b_1 + \dots + \neg b_8 \leq 3$,
- (4) $\neg b_1 + \dots + \neg b_8 \leq 4$.

However, the first three cores are over disjoint sets of soft clauses. PM2 would instead use a set of cardinality constraints in `card_layer` during each iteration:

- (1) $\neg b_1 + \dots + \neg b_3 \leq 1$,
- (2) $\neg b_1 + \neg b_2 + \neg b_3 \leq 1$ **and** $\neg b_4 + \neg b_5 + \neg b_6 \leq 1$,
- (3) $\neg b_1 + \neg b_2 + \neg b_3 \leq 1$ **and** $\neg b_4 + \neg b_5 + \neg b_6 \leq 1$ **and** $\neg b_7 + b_8 \leq 1$.

However, κ_4 is over soft clauses contained in κ_1 and κ_2 . So as to ensure that its cardinality constraints are each over disjoint sets of soft clauses, PM2 would now merge the soft clauses in κ_1 , κ_2 and κ_4 and construct one cardinality constraint over the union of their soft clauses with a bound of 3 (since 3 cores have been discovered over this subset of soft clauses). Hence, at iteration 4, PM2 would use the following set of cardinality constraints in `card_layer`: (4) $\neg b_1 + \dots + \neg b_6 \leq 3$ **and** $\neg b_7 + \neg b_8 \leq 1$.

By splitting up the cores into multiple cardinality constraints PM2 is potentially giving the SAT solver an easier problem to solve. In particular, there are fewer settings of the b_i variables satisfying the two cardinality constraints $\neg b_1 + \dots + \neg b_6 \leq 3$ **and** $\neg b_7 + \neg b_8 \leq 1$ than there are settings satisfying the single cardinality constraint $\neg b_1 + \dots + \neg b_8 \leq 4$.

PM2 also added \geq cardinality constraints (At-Least) to the SAT solver. However, these constraints are logically redundant—the SAT solver has already inferred that at least this number of soft clauses must be falsified. Potentially, these At-Least constraints might slow down the SAT solver.

24.4.5.4. OLL for Weighted MaxSAT

Another algorithm that is very efficient for MaxSAT is the OLL algorithm. OLL is originally an algorithm for optimization in Answer Set Programming [AKMS12] and was adapted for MaxSAT by Morgado et al. [MDM14]. The OLL MaxSAT algorithm was most recently (at the time this chapter was written) implemented in the RC2 solver.

The key idea of OLL is to utilize what are called “soft cardinality constraints.” Consider an unweighted instance for which a core $\kappa = \{\neg b_1, \dots, \neg b_n\}$ has been found (i.e., for each $\neg b_i \in \kappa$, (b_i) is a soft clause with $w((b_i)) = 1$). The formula can be relaxed by removing the b_i from the assumptions and adding a cardinality constraint $\sum_{\neg b_i \in \kappa} \neg b_i < j$. This allows the SAT solver to falsify up to $j-1$ of these soft clauses. Now define new summation output variables o_j for $j = 1, \dots, n-1$ such that o_j is *true* if and only if $\sum_{\neg b_i \in \kappa} \neg b_i \geq j$. That is, o_j is *true* if at least j of the soft clauses $(b_1), \dots, (b_n)$ are falsified; and *false* if at most $j-1$ are falsified.

Algorithm 5: OLL algorithm for MaxSAT [MDM14]

```

1  OLL( $F$ )
2   $F^b = \text{bv\_transform}(F)$ ;  $\text{card\_layer} = \{\}$ ;  $\text{assumptions} = \{b_i \mid (b_i) \in \text{soft}(F^b)\}$ 
3   $F' = \text{hard}(F^b)$ ;  $\text{cost} = \ ; \text{iter} = 0$ ;  $\text{sat?} = \text{false}$ 
4  while not sat? do
    /* SAT solver interface is the same as in Algorithm 2 */
5     $(\text{sat?}, \pi, \kappa) = \text{SATSolve}(F' \cup \text{card\_layer}, \text{assumptions})$ 
6    if not sat? then
7       $wt_{\min} = \min_{\{\ell \mid \ell \in \kappa\}} wt((-\ell))$ 
8       $\text{cost} = \text{cost} + wt_{\min}$ 
9      for  $\ell \in \kappa$  do /*  $(-\ell)$  is a soft clause */
10       |  $wt((-\ell)) - wt_{\min}$ 
11       | if  $wt((-\ell))$  then
12         |  $\text{assumptions} = \text{assumptions} \setminus \{-\ell\}$ 
13       for  $o_j^i \in \kappa$  do /* Summation output variables */
14         | if  $wt((-\neg o_j^i)) \wedge o_{j+1}^i$  exists then
15           | /*  $i$ 'th summation has another output variable */
16           |  $\text{assumptions} = \text{assumptions} \cup \{-o_{j+1}^i\}$ 
17         /* Encode clauses that make output variables equal to value of sum */
18          $\text{card\_layer} = \text{card\_layer} \cup \text{CN}(\sum_{\ell \in \kappa} \ell \geq j \equiv o_j^{\text{iter}} \text{ for } o_2^{\text{iter}}, \dots, o_{|\kappa|}^{\text{iter}})$ 
19         for  $j = 1, \dots, |\kappa|$  do
20           |  $wt((-\neg o_j^{\text{iter}})) - wt_{\min}$ 
21           |  $\text{assumptions} = \text{assumptions} \cup \{-o_j^{\text{iter}}\}$ 
22           |  $\text{iter} = \text{iter} + 1$ 
23     else
24       return  $(\pi|_{\text{vars}(F)}, \text{cost})$ 

```

We will refer to the o_j as being *output* variables of the cardinality constraint, and the literals being summed (the left-hand side) as the *input* variables.

Instead of adding the cardinality constraint $\sum_{-b_i \in \kappa} -b_i < j$, we can add the clauses specifying that for each output variable o_j the equivalence $o_j \equiv \sum_{-b_i \in \kappa} -b_j \geq j$ holds, and call the SAT solver under the assumption $\neg o_j$ (Section 24.4.2.1). This technique is used by [MJML14b] to make MSU3 incremental. However, the use of soft cardinality constraints takes this idea further. Instead of treating the literals $\neg o_j$ as hard assumptions, we can regard them to be new soft clauses $(\neg o_j)_1$. Since $o_j \rightarrow o_{j-1}$, we observe that any truth assignment falsifying the soft clause $(\neg o_j)_1$ must also falsify the j soft clauses $\{(\neg o_1)_1, \dots, (\neg o_j)_1\}$. Hence any feasible solution falsifying $(\neg o_j)_1$ incurs cost j , and this is exactly the same as the costs incurred by falsifying j of the original soft clauses (b_i) (o_j is *true* implies j of the $-b_i$ are *true*). That is, replacing the soft clauses $(b_i)_1$ by the soft clauses $(\neg o_j)_1$ corresponding to the sums of the $-b_i$ is a cost-preserving transformation of the MaxSAT formula.

The main advantage of treating the $(\neg o_i)$ as new soft clauses is that cores can now be found over these variables. For example, the SAT solver could compute a core $\kappa = \{-b_2, -b_4, o_2^2\}$ where $(b_2)_1$ and $(b_4)_1$ are soft clauses of F^b , and o_2^2 is the 2nd output of the second cardinality constraint. This core asserts that either one of $(b_2)_1$ or $(b_4)_1$ must be falsified *or* more than two soft clause inputs of the second cardinality constraint must be falsified. This is clearly a more general

and potentially more powerful type of core than cores strictly over the original soft clauses (b_i). Furthermore, the outputs of one cardinality constraint (the right-hand side variables) can end up being used as the inputs (the left-hand side literals) to additional cardinality constraints, building up new output variables representing quite complicated constraints on the set of original soft clauses that can be falsified.

Algorithm 5 gives the details of the weighted case.⁶ One main feature of this algorithm is that the totalizer encoding [BB03] of cardinality constraints already provides the summation output variables o_i used on line 16 (see Section 24.4.2.1).⁷ Another feature of the algorithm is that through its use of assumptions it can perform clause cloning without having to actually duplicate the clauses as was needed in WPM1/WMSU1 (Section 24.4.5.2); this is described below.

The algorithm starts as normal with an empty `card_layer` and assumptions that force the satisfaction of all soft clauses. Whenever a core is found, the minimum weight over the soft clauses in the core is subtracted from the weight of every soft clause in the core. Note that with summation output literals $\neg o_j^i$ being assumed (lines 15 and 19) the core can contain either negated blocking variables $\neg b_i$ or positive output variables o_j^i . In both cases the corresponding soft clause contains the negation, (b_i) or ($\neg o_j^i$). Line 10 reduces the weight of these soft clauses by w_{\min} . If the weight drops to zero, then that particular soft clause has been fully absorbed into the `card_layer` (i.e., its weight has been fully accounted for in the cost of the cores found so far). In that case we can remove it from the assumptions; otherwise, we continue to assume it but now with reduced weight.

The summation output variables appearing in the core require special treatment if their weight drops to zero. If the weight of o_j^i drops to zero, the i -th cardinality constraint has had j of its inputs made *true*. This corresponds to j soft clauses being falsified, and the weight of these j falsifications has been fully absorbed into `card_layer`. Now we must disallow any further falsifications by assuming $\neg o_{j+1}^i$ (line 15). If further falsifications are needed, the SAT solver will subsequently generate cores containing o_{j+1}^i ; eventually the weight of a $(j+1)$ 'th falsification, i.e., $w(\neg o_{j+1}^i)$, will be absorbed into the `card_layer` and we will then assume o_{j+2}^i . Note that each of the soft clauses ($\neg o_j^i$) has a weight equal to the minimum weight of the soft clauses in the i 'th core (line 18). This is in general different from w_{\min} of the $iter$ 'th core currently being processed. If all of the summation output variables have been absorbed (i.e., there is no o_{j+1}^i output variable) then no further assumptions need be added to the SAT solver—the possible falsification of all of the soft clause inputs to this cardinality constraint has been fully accounted for by the algorithm.

After updating the weights of the soft clauses a new cardinality constraint is generated over the literals in the core (line 16) with a corresponding new set of summation output variables o_j^{iter} . We do not need o_1^{iter} because we know that at

⁶The original publication [MDM14] did not detail the general weighted case, so this algorithm is based on the RC2 implementation.

⁷Algorithms that use the cardinality constraint CN $(\sum_{-b_i \in \text{inCard}} \neg b_i \leq \text{cost})$ with a specific bound, such as MSU3, often use the totalizer encoding that also supplies these summation output variables o_i . To impose the bound of $\leq \text{cost}$, these algorithms add $\neg o_{\text{cost}+1}$ to the set of SAT solver assumptions.

least one of the literals in κ must be *true*, κ specifies a conflict clause satisfied by all feasible solutions (see Section 24.4.1.2), and so o_1^{iter} is already *true*. These new summation output variables will count up how many literals $\neg\ell$ of κ are *true*, and thus how many of the corresponding soft clauses (ℓ) are *false*. Since each of the summation output variables gets weight w_{\min} (line 18), each input clause (ℓ) contributes w_{\min} if falsified. This is how clause cloning is implemented. If the soft clause (ℓ) (for $\neg\ell \in \kappa$) has weight $w(\ell) > w_{\min}$ we will have ℓ as an assumption with new weight $w(\ell) - w_{\min}$, and ℓ as an input to the cardinality constraint with weight w_{\min} . These two uses of the literal ℓ correspond to the two clause copies used in WPM1, and there is no longer a need to add clause copies to the SAT solver.⁸

There are a number of options with respect to assuming the new output variables. In Algorithm 5 the assumptions contain at most one output literal from each cardinality constraint at any time: initially we add $\neg o_2^i$ (line 19), and whenever $w(\neg o_j^i)$ drops to zero we remove it and add the next output literal $\neg o_{j+1}^i$. However, since $\neg o_j^i \rightarrow \neg o_{j+1}^i$ (equivalently $o_{j+1}^i \rightarrow o_j^i$) it is logically sound to assume all of the output literals $\neg o_j^i$ at once [ADR15]. In the RC2 implementation (see [BJM18]) only $w(\neg o_2^{\text{iter}})$ is initialized to w_{\min} (as in line 18). Then, whenever weight is removed from $(\neg o_j^i)$, an equivalent weight is added to $(\neg o_{j+1}^i)$. Thus by the time $(\neg o_2^{\text{iter}})$ has had its weight reduced to zero, $(\neg o_3^{\text{iter}})$ will have had a total of w_{\min} weight added to it. That is, instead of starting $(\neg o_3^{\text{iter}})$ with weight w_{\min} as is done on line 18, weight w_{\min} is moved to this soft clause incrementally. Hence, by the time $(\neg o_j^i), \dots, (\neg o_2^i)$ all have weight zero, $w_{\min} \cdot j$ weight must have been accumulated into `card_layer`.

With multiple output literals o_j^i in the assumptions, the cores (which are not minimal) can contain multiple output literals from the same cardinality constraint. Furthermore, the weight of o_j^i might become zero even when o_{j-1}^i still has positive weight. Hence, solvers such as RC2 might remove o_j^i from the assumptions when its weight gets reduced to zero, and then add it back in when weight is later shifted in from o_{j-1}^i . The impact of this on the efficiency of the algorithm has not been reported to date.

Finally, it can be noted that the cardinality constraint built on line 16 of Algorithm 5 need not be built all at once. Instead, solvers like RC2 build this cardinality constraint incrementally, adding output variables o_j^{iter} only as these are needed.

24.4.5.5. Other Soft Cardinality Constraint Algorithms

Besides OLL other published algorithms have used the notion of dynamically creating new soft clauses representing summations or disjunctions of the original soft clauses. These include, e.g., PMRes [NB14], Maxino [ADR15], and WPM3 [ADG15], all of which have very good performance. We briefly describe their main features, leaving the interested reader to consult the references for more detailed information.

⁸This can yield a significant improvement in efficiency when the core contains hundreds or thousands of soft clauses.

All of these algorithms use the same main loop as Algorithm 5 except that they process cores differently. That is, initially all soft clauses are assumed to be *true* and `card_layer` is empty. Then, for every core κ that is found, w_{\min} is determined and subtracted from every soft clause in κ (line 10). If the remaining weight is greater than zero, a “copy” of the soft clause is left in the assumptions; otherwise, the clause is removed (line 12). This leaves κ with soft clauses all with the same weight w_{\min} .

After this, the core is processed, updating `card_layer` and assumptions. In Algorithm 5 this is done on lines 13 to 19. It is this processing and these lines that are changed by the different algorithms we discuss here.

First, it is useful to describe the properties processing a core must satisfy. Core processing has to update the formula so that the SAT solver is now allowed to falsify *one* of the weight w_{\min} soft clauses in κ : w_{\min} has been added to cost (line 8) so the cost of a single falsification has already been accounted for. Furthermore, the update must preserve the condition that if a truth assignment falsifies j soft clauses of κ , then that truth assignment must incur a cost of $(j - 1) \cdot w_{\min}$ in the new formula.

In OLL this is accomplished by defining the summation output variables o_j^{iter} for $j = 2$ to $|\kappa|$ with totalizer clauses (added to `card_layer`) ensuring that $o_j^{\text{iter}} \equiv \sum_{\ell \in \kappa} \ell \geq j$. Then the soft clauses $(\neg o_j^{\text{iter}})_{w_{\min}}$ are added by adding $\neg o_j^{\text{iter}}$ to the assumptions and setting $w((\neg o_j^{\text{iter}})) = w_{\min}$ (line 18). These soft clauses are added incrementally to the assumptions (line 15), with initially only $\neg o_2^{\text{iter}}$ in the assumptions (line 19). Thus, since $\neg o_1^{\text{iter}}$ is never assumed nor even defined, the SAT solver is allowed to falsify any single soft clause in κ . Furthermore, if a truth assignment falsifies j soft clauses in κ , the variables $o_2^{\text{iter}}, \dots, o_j^{\text{iter}}$ will all be made *true*, and thus $j - 1$ soft clauses $(\neg o_2^{\text{iter}})_{w_{\min}}, \dots, (\neg o_j^{\text{iter}})_{w_{\min}}$ will be falsified incurring cost $(j - 1) \cdot w_{\min}$.

PMRes. The PMRes algorithm processes cores differently. In particular, PMRes does not use summation output variables. Instead it operates as follows. Let $\kappa = \{\neg \ell_1, \dots, \neg \ell_m\}$ and $w = w_{\min}$. Thus the soft clauses are $(\ell_1)_w, \dots, (\ell_m)_w$. PMRes will define $m - 1$ variables d_2, \dots, d_m such that $d_i \equiv \ell_i \vee (\ell_1 \wedge \dots \wedge \ell_{i-1})$, adding the clauses defining this equivalence to `card_layer` ([NB14] presents a compact way of encoding these equivalences). That is, each d_i is *true* if either the soft clause $(\ell_i)_w$ is satisfied or if all prior soft clauses in the sequence $(\ell_1)_w, \dots, (\ell_{i-1})_w$ are satisfied. It will then add the soft clauses $(d_2)_w, \dots, (d_m)_w$ by adding d_2, \dots, d_m to the assumptions and setting $w((d_1)) = w, \dots, w((d_m)) = w$.

In the new formula, the SAT solver is now allowed to falsify any single soft clause in κ . If exactly one $(\ell_i)_w$ is falsified, then all of the d_j assumptions will be *true*: for $j \neq i$, ℓ_j is *true*, making d_j *true*, and for $j = i$, the conjunction $(\ell_1 \wedge \dots \wedge \ell_{i-1})$ is *true*, making d_j *true*. Furthermore, if a truth assignment falsifies j of the soft clauses $(\ell_i)_w$, then it will make d_i *true* for all i corresponding to satisfied soft clauses; it will make d_i *true* for i equal to the index of the first falsified soft since all previous soft clauses are satisfied; and it will make d_i *false* for i equal to the index of all $j - 1$ subsequent falsified soft clauses, since in this case all previous soft clauses are not satisfied. Hence, the truth assignment will falsify exactly $j - 1$ of the soft clauses $(d_i)_w$ in the new formula incurring cost $(j - 1) \cdot w_{\min}$.

PMRes has proved to be a very effective algorithm, with the Eva solver implementing it was one of the best performers in the 2014 MaxSAT evaluation. An open issue with PMRes is that the order of the soft clauses in κ has an influence on the algorithm's performance. Why this is the case is not well understood. A better understanding of this issue might lead to improved ordering schemes and better performance.

Maxino. The Maxino algorithm uses summation output variables to define the new soft clauses like OLL. Its main innovation is to use a simple technique to partition the core κ into a sequence of smaller subsets that are pairwise disjoint and whose union is κ . The algorithm then constructs a sequence of cardinality constraints, each cardinality constraint summing the falsified soft clauses of one partition. It links the cardinality constraints together by including the 1 output (the output variable indicating that the sum of the inputs is ≥ 1) from the previous cardinality constraint as an input of the subsequent cardinality constraint.

Each cardinality constraint allows at most one of its inputs to be *true* (at most one of its corresponding soft clauses to be *false*). In particular, its outputs indicating that the sum is ≥ 2 , ≥ 3 , ..., are all assumed to be *false* and each is given weight w_{\min} (i.e., the negation of these output variables become new soft clauses as in OLL⁹). Its 1 output indicating that the sum is ≥ 1 is not made into a soft clause, instead the 1 output is fed into the subsequent cardinality constraint.

With these new assumptions and clauses capturing the new sequence of cardinality constraints, the SAT solver is now allowed to falsify any single soft clause in κ . Say that the single falsified soft clause is an input to the i 'th cardinality constraint in the sequence of cardinality constraints. The first to $i(-1)$ 'th cardinality constraints will all have only *false* inputs and thus will set all of their outputs to *false*, satisfying the assumptions. The i 'th cardinality constraint has one *true* input, the input corresponding to the falsified soft clause, and will set its 1 output to *true* and all other outputs to *false*. This again satisfies all of the assumptions since the 1-output has not been added to the assumptions. The $(i+1)$ 'th cardinality constraint will have a *true* input, the input corresponding to the 1 output of the i 'th cardinality constraint, and will set its 1 output to *true* and all other outputs to *false*. Again this will satisfy all of the assumptions since the 1 output has not been added to the assumptions. This pattern will continue for the subsequent cardinality constraints and all assumptions will be satisfied.

Furthermore, if j soft clauses in κ are falsified by some truth assignment, then the sequence of cardinality constraints will have j *true* input variables. The first cardinality constraint in the sequence with $i > 0$ *true* inputs, will set its i , $i - 1$, ..., 1 outputs to *true* incurring cost $(i - 1) \cdot w_{\min}$: the 1 output is not a soft clause and hence does not incur any cost while the other outputs all falsify soft clauses with weight w_{\min} . Every subsequent cardinality constraint with k *true* input variables will also get an additional *true* input variable from the previous cardinality constraint's 1 output. Thus, it will set its $k + 1$, k , ..., 1 outputs to *true*

⁹However, all output variables are assumed to be *false*, whereas in OLL only the \geq output is assumed to be *false*. As already explained, OLL incrementally assumes the output variables of the cardinality constraint whereas Maxino assumes them all at once.

incurring cost $k \cdot w_{\min}$: again the 1 output does not incur any cost. Summing over all the cardinality constraints it can be seen that a total cost of $(j-1) \cdot w_{\min}$ will be incurred.

The interested reader may consult [ADR15] for further details. However, we note that in [ADR15] the totalizer encoding is presented in a non-standard way. In particular, for a totalizer with inputs ℓ_1, \dots, ℓ_n and outputs $o_i \equiv \sum_{j=1}^n \ell_j \geq i$ ($1 \leq i \leq n$), [ADR15] uses the non-standard but equivalent encoding $\text{CNF}(\neg \ell_1 + \dots + \neg \ell_n + o_1 + \dots + o_n \geq n)$ conjoined with $\bigwedge_{i=1}^n o_i \rightarrow o_i$. That is, if k inputs ℓ_i are *true* (i.e. at most $k-n$ of the $\neg \ell_i$ are *true*), then at least k outputs must be *true* with the *true* outputs clustered at the lowest indices.

Maxino is another very effective algorithm being one of the best performing algorithms in the 2018 MaxSAT evaluation. An open issue with Maxino is understanding more precisely how partitioning the core into smaller subsets helps the solver, and how this clustering should be done in order to maximize solver performance.

WPM3. The WPM3 algorithm, like Maxino and OLL, uses summation output variables to define the new soft clauses, but it differs in some ways. First, the cardinality constraints it uses are always summations of falsified input soft clauses. The new “meta” soft clauses arising from the output of previous cardinality constraints are not used as inputs to new cardinality constraints as in Maxino, OLL, and PMRes. Second, WPM3 utilizes a sub-optimization step to find the best bound for each core.

When processing a new core κ , WPM, like OLL, distinguishes between the soft clauses that are input soft clauses and those that are output variables of previously constructed cardinality constraints. In particular, if $o_i^j \in \kappa$ is the i 'th output of the j 'th cardinality constraint ($(\neg o_i^j)$ is a soft clause), then WPM3 replaces o_i^j with the union of the inputs of the j 'th cardinality constraint. Since this replacement is done before the new cardinality constraint over the literals in the core is constructed, every cardinality constraint will have only input soft clauses as input (i.e., literals indicating the falsification of some input soft clause). After building this larger core κ , WPM3 constructs a new cardinality constraint C with the literals in κ as inputs and a corresponding set of output variables o_i^C counting the number of falsified soft clauses in κ .

Now instead of assuming the 2 output of the new cardinality constraint C to limit the number of falsified soft clauses to one, WPM3 does a sub-optimization to determine if this limit must be higher. This can be done in various ways, but WPM3 applies the LSU (Algorithm 1) over the subproblem consisting of $h \text{ rd}(F_b)$ along with only the soft clauses that are inputs to C . Say that this sub-optimization concludes that k of the soft clause inputs of C must be falsified. Then WPM3 will add the output $\neg o_{k+1}^C$ to the set of assumptions with weight w_{\min} and will add $(k-1) \cdot w_{\min}$ to the cost. In other words, after concluding that k soft clauses inputs to C must be falsified, WPM3 will construct a problem in which the SAT solver can falsify no more than k of these soft clauses without incurring an additional cost.

24.4.5.5.1. Remarks. The algorithms mentioned in this section, all of which use the technique of adding new soft clauses defined in terms of previous soft clauses, seem to have similar levels of excellent performance. It is not clear if any single approach is generally the best. In particular, much of the demonstrated performance differences seem to arise from the quality of implementation and additional techniques used in the implementation. We close this section by mentioning some of the key techniques used to improve the base algorithm.

An important technique for all core-guided algorithms is weight stratification [ABL09]. This is the technique of partitioning the soft clauses into k sets so that the soft clauses in set i have higher weight than those in set $i - 1$. Then the problem is solved ignoring all soft clauses except those in the k 'th set. Once a solution is found for these higher-weight soft clauses, the $k - 1$ 'th set of soft clauses is added and the problem solved again. This is repeated until the problem has been solved with all soft clauses.

The reason this technique is useful for core-guided algorithms is that it generates cores with higher minimum weight. This moves the cost towards the optimum more rapidly and reduces the amount of clause cloning that needs to occur. In particular, even with clause cloning optimized to be simply reusing a literal (as described earlier), excessive cloning will still slow the solver down by forcing it to generate more cores. For example, if an instance contains a single soft clause of weight 100 that must be falsified in any optimal solution, along with many other soft clauses of weight 1, then the solver would have to generate at least 100 cores if each of these cores has w_{\min} of 1. One hundred such cores would be required to reduce the weight 100 soft clause down to weight zero: before its weight is reduced to zero it remains in the assumptions and must continue to be satisfied by the SAT solver. Furthermore, every core generated augments the complexity of `card_layer`, which starts to slow the SAT solver down. Even with stratification, however, core-guided algorithms become significantly less efficient as the number of distinct soft clause weights increases.

The WPM3 technique of finding an optimal lower bound for each cardinality constraint via a sub-optimization also seems to be quite effective. For example, RC2 implementation of OLL (see [BJM18]) employed this technique (its “`exhaust_core`” method). As pointed out earlier, each added cardinality constraint slows down the SAT solver. The sub-optimization technique allows the solver to increase the cost towards the optimum without adding additional cardinality constraints; optimization is done by using the output variables of the already added cardinality constraint as SAT solver assumptions (recall Section 24.4.5.5).

The final technique that we will mention here is that of core minimization. There are a number of algorithms for reducing the size of a core. For example, one simple technique is as follows. Given a core $\kappa = \{\ell_1, \dots, \ell_n\}$, we can ask the SAT solver if it can satisfy $\neg\ell_1, \dots, \neg\ell_{n-1}$ using assumptions. In the negative case, the SAT solver it will provide a new core that must be a strict subset of κ . In the positive case, the test can be performed leaving out a different literal ℓ_i , $i \neq n$. Depending on the size of κ and the complexity of the current SAT formula (which is growing as we process more cores and add more cardinality constraints), this technique can be effective in reducing the size of the core after which the reduced core can be processed. Core reduction if applied judiciously

can speed up the solver.

Besides the algorithms already mentioned, it should also be noted that a number of other core-guided MaxSAT algorithms have been proposed, e.g., [IMM⁺14, MHM12, MM08, ABL10a]. These algorithms have mostly been subsumed by the algorithms discussed, but some ideas and insights might still be available by examining these earlier works more carefully.

24.4.6. The Implicit Hitting Set Approach

Implicit hitting set (IHS) MaxSAT solvers [DB11, DB13a, DB13b, Dav13, SBJ16] instantiate the implicit hitting set paradigm [DCB10, Kar10] in the context of MaxSAT. In particular, similarly to the purely SAT-based core-guided approach, IHS solvers extract cores in an iterative fashion. However, in contrast to the core-guided approach, in the IHS approach hitting sets over the iteratively accumulated set of cores are computed. If a minimum-cost hitting set (MCHS)¹⁰ of the accumulated cores, when removed from the input formula, makes the formula satisfiable, then any satisfying truth assignment must be an optimal solution [DB11]. In contrast to solvers implementing the purely SAT-based core-guided approach, IHS solvers do not transform the input instance using core compilation steps; rather, the input MaxSAT instance is not altered during search. Each SAT solver call is made on the original hard clauses together with a subset of the original soft clauses (selected through the use of assumptions). Thus the MaxSAT instance does not get larger in size as the search progresses. As a result, the cores found during search remain relatively small compared to the core-guided approaches (consisting of no more than a few hundred clauses on current typical MaxSAT benchmarks).

To obtain a MCHS, current IHS solvers for MaxSAT have employed integer programming (IP) solvers on the following MCHS IP formulation:

$$\begin{aligned}
 & \text{minimize} && \sum_{(b_i) \in \text{soft}(F^b)} w((b_i)) \cdot -b_i \\
 & \text{subject to} && \sum_{-b_i \in \kappa} -b_i \geq 1 && \forall \kappa \in \mathcal{K}, \\
 & && -b_i \in \{0, 1\} && \forall (b_i) \in o(F^b),
 \end{aligned}$$

where \mathcal{K} is the accumulated set of cores, the b_i are the blocking variables for the input soft clauses c_i created by the blocking variable transformation (Section 24.4.1.1.1), and $-b_i = 1$ when b_i is *false* and c_i is falsified.¹¹ In the original

¹⁰Given a set of cores, a hitting set of the cores is a set of soft clauses that includes a soft clause from each core. A hitting set is optimal (of minimum cost) iff the sum of the weights of the soft clauses in it is smallest among all hitting sets of the set of cores.

¹¹Other techniques for computing minimum-cost hitting sets besides using IP solvers can be used. These include branch-and-bound methods [Kle11] and SAT based methods, e.g., [IPLM15]. In certain contexts these methods can be quite effective, e.g., when recomputing a minimum-cost hitting set after adding a single new set. However, these alternatives to IP solvers either cannot handle the weighted case or are not very efficient on the weighted case. In addition, IP solvers support adding other types of constraints that can be extracted from the input formula [DB13a], and bounding techniques that can be used to harden some of the soft clauses [BHJS17]; both of these ideas can yield significant efficiency improvements.

Algorithm 6: IHS algorithm for MaxSAT [DB13b]

```

1 IHS( $F$ )
2  $F^b = \text{bv\_transform}(F)$ ;  $F'$    $\text{hard}(F^b)$ ;  $\mathcal{K} = \{\}$ ;  $HS = \{\}$ 
  /* Initialize upper and lower bounds ( $UB$  and  $LB$ ).  $LB$  is initialized to zero.  $UB$  is
   initialized to a feasible solution. A feasible solution must exist since by assumption
    $\text{hard}(F)$  is satisfiable. Remember the solution yielding the  $UB$  cost. SAT solver
   interface is the same as in Algorithm 2 */
3 ( $\text{sat?}$ ,  $\pi$ ,  $\kappa$ ) = SATSolve ( $F'$ ,  $\{\}$ )
4  $UB = \text{cost}(\pi)$ ;  $LB = 0$ ;  $\text{best\_sol} = \pi$ 
5 while true do
6   ( $\text{sat?}$ ,  $\pi$ ,  $\kappa$ ) = SATSolve ( $F'$ ,  $\{b_i \mid (b_i) \in (\text{soft}(F^b) \setminus HS)\}$ )
7   if not  $\text{sat?}$  then
8      $\mathcal{K} = \mathcal{K} \cup \{\kappa\}$ 
9     ( $HS$ ,  $MCHS?$ ) = ComputeHittingSet( $\mathcal{K}$ )
10    if  $MCHS?$  then
11       $LB = a(LB, \text{cost}(HS))$ 
12      if  $LB \geq UB$  then
13        return ( $\text{best\_sol}$ ,  $\text{cost}(\text{best\_sol})$ )
14    else
15      if  $\text{cost}(\pi) < UB$  then
16         $UB = \text{cost}(\pi)$ 
17         $\text{best\_sol} = \pi$ 
18        if  $LB \geq UB$  then
19          return ( $\text{best\_sol}$ ,  $\text{cost}(\text{best\_sol})$ )

```

IHS algorithm for MaxSAT [DB11] each iteration of the algorithm involved computing a MCHS of the current set of cores \mathcal{K} (which is initially empty) and then testing the satisfiability of $\text{hard}(F^b)$ subject to the assumptions $\{b_i \mid (b_i) \in (\text{soft}(F^b) \setminus HS)\}$. That is, a feasible solution satisfying all soft clauses except those in the MCHS must be found. If this formula is satisfiable, the set must hit *all* cores of the instance [Rei87], and the optimality of implies the optimality of the solution [DB11], terminating the MaxSAT search.

One benefit of the IHS approach is the possibility of tightly integrating upper and lower bound information into search. In fact, starting with [DB13b], more recent versions of IHS MaxSAT solvers have used a different approach that is better cast as an algorithm computing an upper and lower bound and stopping when these two bounds meet. This more modern version of the IHS algorithm is shown in Algorithm 6.

The algorithm starts by applying the blocking variable transformation and initializing the set of cores \mathcal{K} and the hitting set to be empty. It also initializes the lower bound (LB) to be zero and finds a feasible solution whose cost is used to initialize the upper bound (UB). Note that by calling the SAT solver with an empty set of assumptions (line 3) the SAT solver is free to falsify any of the soft clauses. Hence, this call will return SAT if any feasible solution exists, and since by assumption the input set of hard clauses, $\text{hard}(F)$ are satisfiable, this call must return SAT (Section 24.4.1.1.1). The feasible solution is stored in `best_sol` so that an optimal solution can be returned on termination.

Then the main loop is entered where the SAT solver is asked to find a solution satisfying all soft clauses except those in the current hitting set (line 6). If the

solver reports unsatisfiability, the found core κ is added to the set of cores \mathcal{K} , and a new hitting set is computed. If that hitting set is an MCHS, the lower bound can be updated to be the maximum of its previous value and the cost of the MCHS (line 11). In particular, every truth assignment π must falsify at least one soft clause in every core. Thus the set of soft clauses falsified by π forms a hitting set of \mathcal{K} . The cost of π is equal to the sum of the weights of these falsified clauses; that is, it is equal to the cost of the hitting set it generates. Thus $o(\pi)$ must be at least as high as the cost of any MCHS. Since this is true for all truth assignments including the optimal solutions when \mathcal{K} is an MCHS, its cost must be a valid lower bound on $o(F)$.

On the other hand, if the SAT solver returns a satisfying truth assignment, we can check the cost of the returned solution π and update the upper bound if we have found a new lower cost solution. In both cases when either the lower or upper bound is updated we check if the bounds now meet. If they do, we return `best_sol`, since π has now been proven to be an optimal solution.

ComputeHittingSet. The one part of Algorithm 6 that is incompletely specified is how `ComputeHittingSet` works. This subroutine must satisfy two properties: (a) it must return a hitting set of \mathcal{K} , and (b) it must always eventually return an MCHS of \mathcal{K} . That is, `ComputeHittingSet` is free to compute and return non-optimal hitting sets (in which case MCHS? is returned as *false*), but there must always exist a future call that will compute and return an MCHS. In fact, the ability to postpone computing MCHS (which is a computationally hard problem) has been shown to be essential for the effectiveness of the IHS approach [DB13b].

It can be shown that as long as `ComputeHittingSet` satisfies these two properties, Algorithm 6 will eventually terminate returning an optimal solution [BHJS17]. Nevertheless, there is still considerable flexibility in how `ComputeHittingSet` works, and although effective implementations have been found, deep insight has not yet been achieved into how to implement this function optimally. However, Saikko [Sai15] did provide useful empirical evidence about different techniques.

In current IHS solvers (MaxHS and LMHS [BJM18]) the following techniques are used to compute hitting sets.

1. The entire new core κ is added to the previously computed hitting set H . Clearly, if H was a hitting set of all previous cores in \mathcal{K} , $H \cup \kappa$ will be a hitting set of $\mathcal{K} \cup \{\kappa\}$. Typically, this technique is used immediately after any other more expensive technique for computing a hitting set is employed, and it is used until the hitting set becomes so large that the SAT solver returns SAT. Saikko [Sai15] showed that employed in this way this technique is empirically effective.
2. A greedy algorithm can be used to compute a new hitting set. The standard greedy heuristic, where the soft clause that hits the most cores for the least weight is continually selected until all cores are hit, is typically used.
3. The IP solver can be used to compute a hitting set but it can be stopped as soon as it finds a feasible solution with a cost lower than the current upper bound. This technique can help limit the time consumed by the IP

solver.

4. Finally the IP solver is used when it is time to compute an MCHS

These techniques are listed in order of increasing computational cost. ComputeHittingSet implementations typically try to repeat the least expensive technique as many times as they can before moving on to a more expensive technique, with the aim of reducing the time required to accumulate a set of cores capable of raising the lower bound. In particular, when the hitting set returned by a technique immediately yields a solution, there is no longer any point in repeating that technique—no more cores can be found using it until some other cores have been found by other techniques. This “repeat the cheapest until nothing new can be computed” method has proved to be quite effective in practice, but other more sophisticated policies might well be even more effective.

Other Techniques. A number of additional techniques are employed by IHS solvers to speed up the algorithm. Here we mention some of the most important ones.

Seeding is the technique of examining the input MaxSAT instance to determine if any constraints can immediately be added to the IP solver. This technique is quite effective and is more fully described in [DB13a].

Core minimization (Section 24.4.5.5.1) is very effective for IHS solvers [DB13a] and is standardly employed. However, as with its use in core-guided algorithms, the resources expended on core minimization must be limited. For example, it would be too expensive to compute the smallest core. Nevertheless, core minimization tends to be easier to perform in IHS solvers than in core-guided solvers. As already mentioned, the cores in IHS solvers tend to be smaller to begin with, and since cardinality constraints are not being added to the formula the SAT solver calls used during minimization are more efficient. In IHS solvers, minimizing the cores results in tighter constraints for the IP solver, allowing it to raise the lower bound with fewer cores.

The final technique that we will mention here is the IP technique of *reduced cost fixing*. IP solvers standardly work by iteratively solving linear (LP) relaxations of the input IP and then refining those relaxations with cuts or by branching. In the context of IHS, the LP relaxation of the minimum-cost hitting set IP provides a lower bound on the value of the MCHS, which in turn is a lower bound on the cost of an optimal solution to the MaxSAT problem. Hence, as proposed in [BHJS17], the LP can be used to obtain a *reduced cost* for every soft clause by computing the reduced costs of the IP variables $-b_i$. This allows the solver to make use of the standard IP technique of reduced cost fixing [Wol98, DFJ54, CJP83, NW99]. Intuitively, the reduced cost of a soft clause specifies a minimum increase in the cost of the LP relaxation that would arise from falsifying (respectively, satisfying) a soft clause which was satisfied (respectively, falsified) in a computed optimal LP solution. If this cost increase makes the lower bound provided by the LP relaxation greater than the current best known MaxSAT upper bound, the clause (i.e., its blocking variable) can be fixed to its status as prescribed by the LP solution. Fixing the values of blocking variables subsequently simplifies both the minimum-cost hitting set computations as well as the SAT solver calls. This use of a proven IP technique to improve IHS solvers

is made possible by their hybrid use of both SAT and IP solvers. Potentially, other ideas from the rich field of IP solving could be discovered to be useful.

24.4.7. Solving MaxSAT Directly via Integer Programming

In addition to developing SAT-based approaches to MaxSAT, integer programming systems can be used to directly solve MaxSAT instances using the standard encoding of MaxSAT to integer programming [GW93]. In more detail, starting with the blocking variable transformation F^b , an IP 0/1 variable x can be defined for each propositional variable x with $\neg x$ translated as $(1 - x)$. Then every clause can be translated into a ≥ 1 linear constraint, and the objective function can be defined to be the minimization of $\sum_{(b_i) \in \text{soft}(F^b)} w((b_i)) \cdot \neg b_i$. As shown in [AG13, DB13a], IP solvers can be quite effective on some MaxSAT instances using this standard IP encoding of MaxSAT. However, in general, their performance is not competitive with MaxSAT-specific algorithms on most instances.

24.5. Further Developments

Finally, we overview further recent developments in MaxSAT solving involving preprocessing, practical incomplete algorithms, algorithm portfolios, partitioning-based, and parallel approaches to MaxSAT solving.

24.5.1. Preprocessing

Preprocessing techniques for SAT provide significant performance improvements to SAT solving, and preprocessing is today a standard part of the SAT solving workflow (see also Chapter 9). In comparison, progress on MaxSAT preprocessing is more recent and preprocessing less routinely applied.

Nevertheless, essentially all major SAT preprocessing techniques can be naturally lifted to MaxSAT. The key to SAT-based preprocessing for MaxSAT is in introducing first blocking variables through the F^b transformation (cf. Section 24.4.1.1.1), turning each soft clause c into the hard clause $c \vee \neg b_C$. The clauses of $h \text{ rd}(F^b)$ resulting from this transformation can then be preprocessed using an SAT preprocessing with one minor but—in terms of correctness for MaxSAT—very important restriction: the elimination of the blocking variables by resolving on the variables (e.g., through bounded variable elimination) must be disallowed, in order to preserve the unsatisfiable core structure of the instance [BJM13, BMM13]. By lifting the notions of resolution asymmetric tautologies [JHB12] from SAT to MaxSAT, a generic proof of correctness for natural MaxSAT liftings of SAT preprocessing techniques was recently provided [BJ19].

An optimal solution to the original MaxSAT instance can then be obtained from any optimal solution to the instance after applying SAT preprocessing by applying the generic solution reconstruction technique for SAT preprocessing [JHB12] in linear time with respect to the number of preprocessing steps made.

After applying SAT preprocessing the clauses can contain more than one

blocking variable, and a blocking variable can occur in multiple clauses.¹² Hence the bijective mapping between blocking variables and soft clauses is lost after SAT preprocessing. For coupling SAT preprocessing with core-guided and IHS MaxSAT approaches, this is not problematic, but nevertheless requires attention. The blocking variables introduced before preprocessing can be directly used as blocking variables by the MaxSAT algorithms which employ blocking variables, instead of performing another F^b transformation step after preprocessing [BSJ15a].

Variables that can be directly used as blocking variables can also be detected from input MaxSAT instances and reused as assumptions throughout the MaxSAT solving process, thereby limiting the number of new variables added by the F^b transformation step, as proposed in [BSJ15b] under the notion of *group detection*. Group detection refers to detecting auxiliary variables introduced by MaxSAT encodings of more high-level finite-domain constraints. In more detail, assume that a set of clauses $\text{CNF}(\mathcal{C}) = \{C_1, \dots, C_k\}$ is a conjunctive normal form encoding of a finite-domain constraint \mathcal{C} . Now assume that \mathcal{C} is a soft constraint, with an associated weight $W_{\mathcal{C}}$. On the level of *group MaxSAT* [HMM15] this soft constraint can be represented as the soft group $\{C_1, \dots, C_k\}$ with weight $W_{\mathcal{C}}$. For employing a standard MaxSAT solver, a natural way of encoding such a group-level MaxSAT representation is to introduce a single blocking variable $b_{\mathcal{C}}$ and add it to each of the clauses C_1, \dots, C_k , thereby obtaining the hard clauses $C_1 \vee \neg b_{\mathcal{C}}, \dots, C_k \vee \neg b_{\mathcal{C}}$ together with the soft clause $b_{\mathcal{C}}$ with weight $W_{\mathcal{C}}$. With this intuition, group detection refers to pattern matching in a MaxSAT instance literals that are directly re-usable as blocking variables. In particular, a literal l is re-usable as a blocking variable if the literal $\neg l$ only appears in the hard clauses of the MaxSAT instance. This captures the auxiliary variables arising from the already-described general approach of encoding high-level soft constraints in MaxSAT. In practice, in [BSJ15b] the authors showed that in the industrial and crafted benchmark instances of MaxSAT Evaluation 2014, variables re-usable directly as blocking variables were found in a majority of the instances using group detection, most often in significant numbers.

The impact of SAT preprocessing on the efficiency of MaxSAT algorithms in practice has been to date less studied, although some theoretical evidence on its potential is available; see e.g. [BMM13, KBSJ17]. However, it has been shown that, theoretically, SAT preprocessing has no effect on the best-case number of iterations of core-guided MaxSAT algorithms [BJ16], which suggests that its impact may mainly be in making the individual SAT solver calls within core-guided MaxSAT algorithms simpler.

Beyond SAT preprocessing, native MaxSAT-specific preprocessing techniques have also been proposed. A limited form of variable saturation in the context of MaxSAT resolution was proposed as a MaxSAT preprocessing technique in [ALM08]. More recently, the MaxPre MaxSAT preprocessor [KBSJ17] introduced several further preprocessing techniques native to MaxSAT, including label matching, binary core removal, and generalized form of subsumed label elimina-

¹²In this context, blocking variables are sometimes referred to as *labels*, following the *labeled CNF* framework [BM12] that explicitly distinguishes between the blocking variables and original clauses of each clause.

tion [BSJ16], the last of which allows for removing blocking variables (labels) in the following sense: A blocking variable b is subsumed by a set of blocking variables $\{b_1, \dots, b_k\}$ if one of b_1, \dots, b_k occurs in a clause whenever b occurs in the clause, and $\sum_{i=1}^k w(b_i) \leq w(b)$. For $k = 1$, this gives the original proposed definition of subsumed label elimination [BSJ16], while in general the conditions correspond to the NP-complete hitting set problem. In practice, an approximate version of the generalized form of subsumed label elimination can be computed using a classical $\ln(n)$ -approximation algorithm for the hitting set problem.

In terms of implementations of MaxSAT preprocessing, the SAT preprocessor Coprocessor [Man12] supports SAT preprocessing techniques for MaxSAT instances, while the MaxPre preprocessor [KBSJ17] offers combinations of SAT preprocessing and native MaxSAT preprocessing techniques.

24.5.2. Incomplete Algorithms for MaxSAT

Incomplete MaxSAT solving can primarily be divided into two categories: (i) local search MaxSAT solvers and (ii) approximation algorithms based on iterative calls to SAT solvers that may not guarantee optimality.

24.5.2.1. Local Search for MaxSAT

Local search solvers start by finding a random assignment to the MaxSAT formula. Since this assignment is unlikely to satisfy all clauses, they choose a clause that is unsatisfied and flip the assignment of a variable occurring in that clause. When compared to local search solvers for SAT, local search solvers for MaxSAT face additional challenges since they must find an assignment that satisfies all hard clauses while trying to maximize the sum of the weights of satisfied soft clauses. To overcome this challenge, local MaxSAT solvers distinguish between soft and hard clauses and apply weighting schemes that favor satisfying hard clauses [CLTS14, LCW⁺15, CJS15, CLLS16, LCSH17, LC18]. For more details on local search SAT solvers, we refer the interested reader to Chapter 6.

24.5.2.2. Model-Improving Approximation

Model-improving approximation algorithms do not guarantee the optimality of the final solution. An example of this approach is an enumeration of minimal correction subsets (MCSes) [MHJ⁺13, MPM15]. An MCS of an unsatisfiable set of clauses is a minimal subset that, if removed, makes the formula satisfiable. There has been a recent trend of using approximation approaches for incomplete MaxSAT. For instance, an approach based on bit-vector optimization was proposed for unweighted incomplete MaxSAT with promising results [Nad18].

LINSBPS [DS18] is an incomplete MaxSAT solver based on model-improving algorithms. For weighted instances, they simplify the weights of the formula and see it in *low resolution*, i.e., with all their weights divided by a large value. Since they use integer division, the weights of some soft clauses are set to zero, effectively removing them. Whenever the solver terminates with an “optimal” solution to the simplified formula, the resolution is increased by decreasing the divisor value. Additionally, they use solution-based saving [AG17, DCS18] to guide the search towards the best solution found so far.

OPEN-WBO-INC [JKMR18] is another incomplete MaxSAT solver based on model-improving algorithms. For weighted instances, this solver also simplifies the weights of the formula by clustering them into k different weights. An alternative approach also employed by this solver is to consider a weighted instance as a multilevel optimization problem (even though it may not satisfy the BMO condition described in Section 24.3.1.1). This alternative approach reduces the solving of a weighted instance to solving a sequence of unweighted instances, which significantly improves the performance of model-improving algorithms and quickly converges to a final solution which corresponds to an upper bound on the optimal value of the original formula.

24.5.3. Algorithm Portfolios

Given the diversity of MaxSAT algorithms (some of which were overviewed in Section 24.4), a meta approach that aims at taking advantage of this diversity is to build an oracle to predict the most suitable MaxSAT algorithm for a particular instance. Inspired by the success of SATzilla [XHH08] and other algorithm portfolios for SAT solving, Matos et al. [MPLM08] proposed the first algorithm portfolio for MaxSAT by employing linear regression using three kinds of features [NLH⁺04]: problem size features, balance features and local search probe features.

ISAC+ [AMS14] is another algorithm portfolio approach that extends the instance-specific algorithm configuration (ISAC [KMST10]) by clustering the benchmarks and tuning the parameters of existing solvers on each cluster. ISAC does not use regression-based analysis but instead computes a representative feature vector that captures properties of a particular instance in order to identify clusters of similar instances. A single solver is selected for each group based on the characteristics of those benchmarks. Given a new instance, its features are computed and it is assigned to the closest cluster and solved by the respective solver. The features used by ISAC+ consider the percentage of clauses that are soft, and the statistics of the distribution of weights (average, minimum, maximum, standard deviation). The remaining 32 features are a subset of the features used by SATZilla that are specific to SAT instances and are applied to MaxSAT instances by considering all soft clauses as hard. ISAC+ differs from ISAC since it also applies an algorithm selector to choose the best parameters of existing solvers for each cluster. Using a portfolio of different MaxSAT solvers, allowed ISAC+ to be one of the best approaches for MaxSAT in the 2013-2015 MaxSAT Evaluations.

24.5.4. Parallel Solving

Nowadays, extra computational power is no longer coming from higher processor frequencies but rather from adding additional processors. Distributed systems are also becoming more predominant and cheaper. For instance, it is not uncommon to deploy distributed tools to web services such as Amazon Web Services. Even though parallel approaches for MaxSAT are not as predominant as in SAT solving, there are a few parallel MaxSAT solvers that can take advantage of these new architectures. These parallel solvers can be mainly categorized into two classes: (i) portfolio approaches, and (ii) search space splitting.

24.5.4.1. Portfolio Approaches

Portfolio approaches are not restricted to sequential solving and can be extended to a parallel setting where different strategies are run in parallel. These different strategies can be based on different MaxSAT algorithms (e.g., model-improving algorithms, core-guided algorithms, implicit hitting set algorithms) or by using different settings of a given algorithm (e.g., changing the encoding for cardinality or pseudo-Boolean constraints). An example of a portfolio solver is PWBO [MML11] which allocates half of the workers to use model-improving algorithms, and the other half core-guided algorithms. To further increase diversity between workers using the same algorithm, PWBO translates cardinality and pseudo-Boolean constraints with different encodings [MML11].

The portfolio approaches for parallel MaxSAT solving are closely related to portfolio approaches for parallel SAT solving. The main difference consists on how diversity is employed. Both approaches perform clause sharing where learned clauses between different workers are exchanged to further prune the search space and boost the performance of the parallel solver.

Portfolio approaches are better suited for multicore approaches since diversity does not usually scale and clause sharing is more efficient when performed in shared memory.

24.5.4.2. Search Space Splitting

Another approach for parallel MaxSAT solving is to split the original problem into subproblems such that each worker solves a smaller formula that is hopefully easier to solve. The two main approaches to divide the search space is to (i) perform interval splitting or (ii) split the search space with guiding paths.

The optimal solution value of a MaxSAT problem is bounded between 0 and the sum of the weights of the soft clauses. Therefore one way of splitting the search space is to divide this interval and have different workers searching on different bounds in order to narrow the search interval and converge to an optimal solution. This approach is employed in a version of PWBO [MML12].

The search space can also be viewed as a binary tree, where each node corresponds to a variable and each of its edges corresponds to an assignment to that variable. The search space can then be split by using different paths in the tree (called guiding paths) to split the search into sub-trees and assign each of them to a different worker. There are many ways of generating guiding paths. One of the most successful approaches has been using lookahead solvers [vdTHB12] to split the search tree. This strategy has been used in a distributed MaxSAT solver [TLM16].

One of the advantages of this approach is that it is easier to split the search space into many workers and therefore can be more suitable for a distributed setting. However, since some of the sub-trees may be much easier to solve than others, it is necessary to have a dynamic load balancing scheme that continuously splits the search space and keeps all workers busy.

24.5.5. Partitioning-based MaxSAT

Divide-and-conquer techniques are not restricted to parallel MaxSAT solving and can also be applied to sequential MaxSAT solving. For instance, partitioning-based MaxSAT has been successfully used to improve the performance of core-based algorithms [MML13, NMJ⁺15].

Core-based algorithms rely on unsatisfiable cores returned by the SAT solver. Usually, core-based algorithms perform better when small cores are found early. However, this is not necessarily the case, since these unsatisfiable cores are not guaranteed to be minimal, and there are no guarantees that smaller cores will be found first. The order in which cores are enumerated can have a large impact on the performance of MaxSAT algorithms.

Similarly to SAT instances, MaxSAT instances can also be represented using graphs. For instance, a MaxSAT instance can be represented by either the variable incidence graph (VIG) or the clause variable incidence graph (CVIG) [MML13]. In the VIG representation, vertices correspond to the variables of the problem, and edges correspond to clauses where both variables occur. In the CVIG representation, vertices correspond to both variables and clauses and there is an edge between the vertices that represent the variables and the clauses where they occur. One can also use the resolution rule to construct a graph where vertices are clauses and edges correspond to clauses that can be resolved with each other (RES) [NMJ⁺15].

Partitioning-based MaxSAT uses graph partition techniques to split the formula into disjoint subsets (e.g., using a modularity clustering approach [GN02]). By decomposing the MaxSAT instance into disjoint subformulas, the SAT solver is forced to find unsatisfiable cores on a subset of the formula. If such unsatisfiable cores exist, then they may be much smaller than if the entire MaxSAT instance was given to the SAT solver. The RES representation [NMJ⁺15] is implemented on top of the Open-WBO system [MML14] using the MSU3 algorithm and has been shown to be particularly effective for unweighted MaxSAT.

24.6. Summary

Advances in practical aspects of maximum satisfiability within the last 10-15 years have established MaxSAT as a general-purpose Boolean constraint optimization paradigm. The improvements have come in increasing numbers both in terms of domain-specific MaxSAT encodings of various hard optimization problems to novel algorithmic techniques and their efficient implementations in general-purpose MaxSAT solvers. In terms of algorithmic advances, the advent of SAT-based approaches to practical MaxSAT solving in its different forms is arguably at the center of this success. To this end, we have here aimed at covering central research advances on practical aspects of MaxSAT within the last 10 years, with a strong focus on the three main SAT-based approaches to MaxSAT solving: the model-improving, core-guided, and the implicit hitting set approach. Beyond further improvements to these algorithmic approaches to MaxSAT, there are plenty of avenues for further research on MaxSAT in terms of novel applications of the

technology, as well as further advances in the recent research directions on incomplete and parallel approaches to MaxSAT solving.

Acknowledgments

The authors thank Jeremias Berg, Alexey Ignatiev, Joao Marques-Silva, and Antonio Morgado, who provided various helpful comments on a draft version of this chapter.

References

- [ABJM17] C. Ansótegui, F. Bacchus, M. Järvisalo, and R. Martins, editors. *MaxSAT Evaluation 2017: Solver and Benchmark Descriptions*, volume B-2017-2 of *Department of Computer Science Series of Publications B*. University of Helsinki, 2017.
- [ABL09] C. Ansótegui, M. L. Bonet, and J. Levy. Solving (weighted) partial MaxSAT through satisfiability testing. In O. Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 427–440. Springer, 2009.
- [ABL10a] C. Ansótegui, M. L. Bonet, and J. Levy. A new algorithm for weighted partial MaxSAT. In M. Fox and D. Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press, 2010.
- [ABL⁺10b] J. Argelich, D. L. Berre, I. Lynce, J. Marques-Silva, and P. Rápicault. Solving linux upgradeability problems using boolean optimization. In I. Lynce and R. Treinen, editors, *Proceedings First International Workshop on Logics for Component Configuration, LoCoCo 2010, Edinburgh, UK, 10th July 2010.*, volume 29 of *EPTCS*, pages 11–22, 2010.
- [ABL13] C. Ansótegui, M. L. Bonet, and J. Levy. SAT-based MaxSAT algorithms. *Artificial Intelligence*, 196:77–105, 2013.
- [ACLM12] J. Argelich, A. Cabiscol, I. Lynce, and F. Manyà. Efficient encodings from CSP into SAT, and from MaxCSP into MaxSAT. *Multiple-Valued Logic and Soft Computing*, 19(1-3):3–23, 2012.
- [ADG15] C. Ansótegui, F. Didier, and J. Gabàs. Exploiting the structure of unsatisfiable cores in MaxSAT. In Q. Yang and M. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 283–289. AAAI Press, 2015.
- [ADR15] M. Alviano, C. Dodaro, and F. Ricca. A MaxSAT algorithm using cardinality constraints of bounded size. In Q. Yang and M. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI*

- 2015, Buenos Aires, Argentina, July 25-31, 2015, pages 2677–2683. AAAI Press, 2015.
- [AG13] C. Ansótegui and J. Gabàs. Solving (weighted) partial MaxSAT with ILP. In C. P. Gomes and M. Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings*, volume 7874 of *Lecture Notes in Computer Science*, pages 403–409. Springer, 2013.
- [AG17] C. Ansótegui and J. Gabàs. WPM3: an (in)complete algorithm for weighted partial MaxSAT. *Artificial Intelligence*, 250:37–57, 2017.
- [AIMT13] C. Ansótegui, I. Izquierdo, F. Manyà, and J. Torres-Jiménez. A Max-SAT-based approach to constructing optimal covering arrays. In K. Gibert, V. J. Botti, and R. R. Bolaño, editors, *Artificial Intelligence Research and Development - Proceedings of the 16th International Conference of the Catalan Association for Artificial Intelligence, Vic, Catalonia, Spain, October 23-25, 2013.*, volume 256 of *Frontiers in Artificial Intelligence and Applications*, pages 51–59. IOS Press, 2013.
- [AKMS12] B. Andres, B. Kaufmann, O. Matheis, and T. Schaub. Unsatisfiability-based optimization in clasp. In A. Dovier and V. S. Costa, editors, *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, September 4-8, 2012, Budapest, Hungary*, volume 17 of *LIPICs*, pages 211–221. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.
- [ALM08] J. Argelich, C. M. Li, and F. Manyà. A preprocessor for MaxSAT solvers. In H. K. Büning and X. Zhao, editors, *Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings*, volume 4996 of *Lecture Notes in Computer Science*, pages 15–20. Springer, 2008.
- [ALM09] J. Argelich, I. Lynce, and J. Marques-Silva. On solving boolean multilevel optimization problems. In C. Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 393–398. AAAI Press, 2009.
- [ALMP08] J. Argelich, C. M. Li, F. Manyà, and J. Planes. The first and second Max-SAT evaluations. *Journal on Satisfiability, Boolean Modeling and Computation*, 4(2-4):251–278, 2008.
- [ALMP11] J. Argelich, C. M. Li, F. Manyà, and J. Planes. Analyzing the instances of the MaxSAT evaluation. In K. A. Sakallah and L. Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, volume 6695 of *Lecture Notes in Computer Science*, pages 360–

361. Springer, 2011.
- [AMS14] C. Ansótegui, Y. Malitsky, and M. Sellmann. MaxSAT by improved instance-specific algorithm configuration. In C. E. Brodley and P. Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pages 2594–2600. AAAI Press, 2014.
- [AN14] R. J. A. Achá and R. Nieuwenhuis. Curriculum-based course timetabling with SAT and MaxSAT. *Annals of Operations Research*, 218(1):71–91, 2014.
- [ANOR09] R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. Cardinality networks and their applications. In O. Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 167–180. Springer, 2009.
- [ARMS02] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. Generic ILP versus specialized 0-1 ILP: an update. In *Proceedings of the 2002 IEEE/ACM International Conference on Computer-aided Design, ICCAD 2002, San Jose, California, USA, November 10-14, 2002*, pages 450–457, 2002.
- [BACF17] A. Belabed, E. Aïmeur, M. A. Chikh, and H. Fethallah. A privacy-preserving approach for composite web service selection. *Transactions on Data Privacy*, 10(2):83–115, 2017.
- [BB03] O. Bailleux and Y. Boufkhad. Efficient CNF encoding of boolean cardinality constraints. In F. Rossi, editor, *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, volume 2833 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2003.
- [BBR09] O. Bailleux, Y. Boufkhad, and O. Roussel. New encodings of pseudo-boolean constraints into CNF. In O. Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 181–194. Springer, 2009.
- [BDS11] P. Bieber, R. Delmas, and C. Seguin. DALculus – theory and tool for development assurance level allocation. In F. Flammini, S. Bologna, and V. Vittorini, editors, *Computer Safety, Reliability, and Security - 30th International Conference, SAFECOMP 2011, Naples, Italy, September 19-22, 2011. Proceedings*, volume 6894 of *Lecture Notes in Computer Science*, pages 43–56. Springer, 2011.
- [BGSV15] M. Boffill, M. Garcia, J. Suy, and M. Villaret. MaxSAT-based scheduling of B2B meetings. In L. Michel, editor, *Integration of AI and OR Techniques in Constraint Programming - 12th*

- International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings*, volume 9075 of *Lecture Notes in Computer Science*, pages 65–73. Springer, 2015.
- [BHJS17] F. Bacchus, A. Hyttinen, M. Järvisalo, and P. Saikko. Reduced cost fixing in MaxSAT. In J. C. Beck, editor, *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10416 of *Lecture Notes in Computer Science*, pages 641–651. Springer, 2017.
- [BJ13] J. Berg and M. Järvisalo. Optimal correlation clustering via MaxSAT. In W. Ding, T. Washio, H. Xiong, G. Karypis, B. M. Thuraisingham, D. J. Cook, and X. Wu, editors, *13th IEEE International Conference on Data Mining Workshops, ICDM Workshops, TX, USA, December 7-10, 2013*, pages 750–757. IEEE Computer Society, 2013.
- [BJ14] J. Berg and M. Järvisalo. SAT-based approaches to treewidth computation: An evaluation. In *26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2014, Limassol, Cyprus, November 10-12, 2014*, pages 328–335. IEEE Computer Society, 2014.
- [BJ16] J. Berg and M. Järvisalo. Impact of SAT-based preprocessing on core-guided MaxSAT solving. In M. Rueher, editor, *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, volume 9892 of *Lecture Notes in Computer Science*, pages 66–85. Springer, 2016.
- [BJ17] J. Berg and M. Järvisalo. Cost-optimal constrained correlation clustering via weighted partial maximum satisfiability. *Artificial Intelligence*, 244:110–142, 2017.
- [BJ19] J. Berg and M. Järvisalo. Unifying reasoning and core-guided search for maximum satisfiability. In F. Calimeri, N. Leone, and M. Manna, editors, *Logics in Artificial Intelligence - 16th European Conference, JELIA 2019, Rende, Italy, May 7-11, 2019, Proceedings*, volume 11468 of *Lecture Notes in Computer Science*, pages 287–303. Springer, 2019.
- [BJB⁺14] K. Bunte, M. Järvisalo, J. Berg, P. Myllymäki, J. Peltonen, and S. Kaski. Optimal neighborhood preserving visualization by maximum satisfiability. In C. E. Brodley and P. Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pages 1694–1700. AAAI Press, 2014.
- [BJM13] A. Belov, M. Järvisalo, and J. Marques-Silva. Formula preprocessing in MUS extraction. In N. Piterman and S. A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24,*

2013. *Proceedings*, volume 7795 of *Lecture Notes in Computer Science*, pages 108–123. Springer, 2013.
- [BJM14] J. Berg, M. Järvisalo, and B. Malone. Learning optimal bounded treewidth Bayesian networks via maximum satisfiability. In J. Corander and S. Kaski, editors, *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics, AISTATS 2014, Reykjavik, Iceland, April 22-25, 2014*, volume 33 of *JMLR Workshop and Conference Proceedings*, pages 86–95. JMLR.org, 2014.
- [BJM18] F. Bacchus, M. Järvisalo, and R. Martins, editors. *MaxSAT Evaluation 2018: Solver and Benchmark Descriptions*, volume B-2018-2 of *Department of Computer Science Series of Publications B*. University of Helsinki, 2018.
- [BJM19] F. Bacchus, M. Järvisalo, and R. Martins. MaxSAT Evaluation 2018: New developments and detailed results. *Journal on Satisfiability, Boolean Modeling and Computation*, 11:99–131, 2019.
- [BLM07] M. L. Bonet, J. Levy, and F. Manyà. Resolution for Max-SAT. *Artificial Intelligence*, 171(8-9):606–618, 2007.
- [BM12] A. Belov and J. Marques-Silva. Generalizing redundancy in propositional logic: Foundations and hitting sets duality. *CoRR*, abs/1207.1257, 2012.
- [BMM13] A. Belov, A. Morgado, and J. Marques-Silva. SAT-based preprocessing for MaxSAT. In K. L. McMillan, A. Middeldorp, and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, volume 8312 of *Lecture Notes in Computer Science*, pages 96–111. Springer, 2013.
- [BP10] D. L. Berre and A. Parrain. The Sat4j library, release 2.2. *Journal of Satisfiability, Boolean Modeling and Computation*, 7(2-3):59–6, 2010.
- [BSJ15a] J. Berg, P. Saikko, and M. Järvisalo. Improving the effectiveness of SAT-based preprocessing for MaxSAT. In Q. Yang and M. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 239–245. AAAI Press, 2015.
- [BSJ15b] J. Berg, P. Saikko, and M. Järvisalo. Re-using auxiliary variables for MaxSAT preprocessing. In *27th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2015, Vietri sul Mare, Italy, November 9-11, 2015*, pages 813–820. IEEE Computer Society, 2015.
- [BSJ16] J. Berg, P. Saikko, and M. Järvisalo. Subsumed label elimination for maximum satisfiability. In G. A. Kaminka, M. Fox, P. Bouquet, E. Hüllermeier, V. Dignum, F. Dignum, and F. van Harmelen, editors, *ECAI 2016 - 22nd European Con-*

ference on Artificial Intelligence, 29 August-2 September 2016, The Hague, The Netherlands - Including Prestigious Applications of Artificial Intelligence (PAIS 2016), volume 285 of *Frontiers in Artificial Intelligence and Applications*, pages 630–638. IOS Press, 2016.

- [CHB17] E. Cohen, G. Huang, and J. C. Beck. (I can get) satisfaction: Preference-based scheduling for concert-goers at multi-venue music festivals. In S. Gaspers and T. Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10491 of *Lecture Notes in Computer Science*, pages 147–163. Springer, 2017.
- [CJP83] H. Crowder, E. L. Johnson, and M. Padberg. Solving large-scale zero-one linear programming problems. *Operations Research*, 31(5):803–834, 1983.
- [CJS15] S. Cai, Z. Jie, and K. Su. An effective variable selection heuristic in SLS for weighted Max-2-SAT. *Journal of Heuristics*, 21(3):433–456, 2015.
- [CLLS16] S. Cai, C. Luo, J. Lin, and K. Su. New local search methods for partial MaxSAT. *Artificial Intelligence*, 240:1–18, 2016.
- [CLTS14] S. Cai, C. Luo, J. Thornton, and K. Su. Tailoring local search for partial MaxSAT. In C. E. Brodley and P. Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pages 2623–2629. AAAI Press, 2014.
- [CSMV10] Y. Chen, S. Safarpour, J. Marques-Silva, and A. G. Veneris. Automated design debugging with maximum satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(11):1804–1817, 2010.
- [CSVMS09] Y. Chen, S. Safarpour, A. G. Veneris, and J. Marques-Silva. Spatial and temporal design debug using partial MaxSAT. In F. Lombardi, S. Bhanja, Y. Massoud, and R. I. Bahar, editors, *Proceedings of the 19th ACM Great Lakes Symposium on VLSI 2009, Boston Area, MA, USA, May 10-12 2009*, pages 345–350. ACM, 2009.
- [Dav13] J. Davies. *Solving MAXSAT by Decoupling Optimization and Satisfaction*. PhD thesis, University of Toronto, 2013.
- [DB11] J. Davies and F. Bacchus. Solving MAXSAT by solving a sequence of simpler SAT instances. In J. H. Lee, editor, *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, volume 6876 of *Lecture Notes in Computer Science*, pages 225–239. Springer, 2011.
- [DB13a] J. Davies and F. Bacchus. Exploiting the power of MIP solvers in MaxSAT. In M. Jarvisalo and A. V. Gelder, editors, *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12,*

2013. *Proceedings*, volume 7962 of *Lecture Notes in Computer Science*, pages 166–181. Springer, 2013.
- [DB13b] J. Davies and F. Bacchus. Postponing optimization to speed up MAXSAT solving. In C. Schulte, editor, *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, volume 8124 of *Lecture Notes in Computer Science*, pages 247–262. Springer, 2013.
- [DCB10] J. Davies, J. Cho, and F. Bacchus. Using learnt clauses in MaxSAT. In D. Cohen, editor, *Principles and Practice of Constraint Programming - CP 2010 - 16th International Conference, CP 2010, St. Andrews, Scotland, UK, September 6-10, 2010. Proceedings*, volume 6308 of *Lecture Notes in Computer Science*, pages 176–190. Springer, 2010.
- [DCS18] E. Demirovic, G. Chu, and P. J. Stuckey. Solution-based phase saving for CP: A value-selection heuristic to simulate local search behavior in complete solvers. In J. N. Hooker, editor, *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, pages 99–108. Springer, 2018.
- [Dec03] R. Dechter. *Constraint processing*. Elsevier Morgan Kaufmann, 2003.
- [DFJ54] G. B. Danzig, D. R. Fulkerson, and S. M. Johnson. Solution of a large-scale traveling-salesman problem. *Operations Research*, 2:393–410, 1954.
- [DGT18] R. Dimitrova, M. Ghasemi, and U. Topcu. Maximum realizability for linear temporal logic specifications. In S. K. Lahiri and C. Wang, editors, *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, volume 11138 of *Lecture Notes in Computer Science*, pages 458–475. Springer, 2018.
- [DM17] E. Demirovic and N. Musliu. MaxSAT-based large neighborhood search for high school timetabling. *Computers & Operations Research*, 78:172–180, 2017.
- [DMW17] E. Demirović, N. Musliu, and F. Winter. Modeling and solving staff scheduling with partial weighted MaxSAT. *Annals of Operations Research*, 2017.
- [dOS15] R. T. de Oliveira and F. Silva. On a relative MaxSAT encoding for the steiner tree problem in graphs. In O. Pichardo-Lagunas, O. Herrera-Alcántara, and G. Arroyo-Figueroa, editors, *Advances in Artificial Intelligence and Its Applications - 14th Mexican International Conference on Artificial Intelligence, MICAI 2015, Cuernavaca, Morelos, Mexico, October 25-31, 2015. Proceedings, Part II*, volume 9414 of *Lecture Notes in Computer Science*, pages 422–434. Springer, 2015.

- [DS18] E. Demirovic and P. J. Stuckey. LinSBPS. In *MaxSAT Evaluation 2018: Solver and Benchmark Descriptions*, volume B-2018-2 of *Department of Computer Science Series of Publications B*, page 8. University of Helsinki, 2018.
- [ER18] R. Ehlers and F. P. Romero. Approximately propagation complete and conflict propagating constraint encodings. In O. Beyersdorff and C. M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10929 of *Lecture Notes in Computer Science*, pages 19–36. Springer, 2018.
- [ES03] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.
- [ES06] N. Eén and N. Sörensson. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, 2006.
- [FBM⁺17] Y. Feng, O. Bastani, R. Martins, I. Dillig, and S. Anand. Automated synthesis of semantic malware signatures using maximum satisfiability. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.
- [FG10] A. M. Frisch and P. A. Giannoros. SAT encodings of the at-most-k constraint. In *Proceedings of 9th International Workshop on Constraint Modelling and Reformulation*, 2010.
- [FLQ⁺14] Z. Fang, C. Li, K. Qiao, X. Feng, and K. Xu. Solving maximum weight clique using maximum satisfiability reasoning. In T. Schaub, G. Friedrich, and B. O’Sullivan, editors, *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 303–308. IOS Press, 2014.
- [FM06] Z. Fu and S. Malik. On solving the partial MAX-SAT problem. In A. Biere and C. P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 of *Lecture Notes in Computer Science*, pages 252–265. Springer, 2006.
- [GL12] J. Guerra and I. Lynce. Reasoning over biological networks using maximum satisfiability. In M. Milano, editor, *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, volume 7514 of *Lecture Notes in Computer Science*, pages 941–956. Springer, 2012.

- [GML11] A. Graça, J. Marques-Silva, and I. Lynce. Haplotype inference using propositional satisfiability. In R. Bruni, editor, *Mathematical Approaches to Polymer Sequence Analysis and Related Problems*, pages 127–147. Springer, 2011.
- [GMLO11] A. Graça, J. Marques-Silva, I. Lynce, and A. L. Oliveira. Haplotype inference with pseudo-boolean optimization. *Annals of Operations Research*, 184(1):137–162, 2011.
- [GN02] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002.
- [GW93] M. X. Goemans and D. P. Williamson. A new $\frac{3}{4}$ -approximation algorithm for MAXSAT. In G. Rinaldi and L. A. Wolsey, editors, *Proceedings of the 3rd Integer Programming and Combinatorial Optimization Conference, Erice, Italy, April 29 - May 1, 1993*, pages 313–321. CIACO, 1993.
- [HKD⁺16] W. Huang, D. A. Kitchaev, S. T. Dacek, Z. Rong, A. Urban, S. Cao, C. Luo, and G. Ceder. Finding and proving the exact ground state of a generalized ising model by convex optimization and MAX-SAT. *Physical Review B*, 94:134424, 2016.
- [HLdGS08] F. Heras, J. Larrosa, S. de Givry, and T. Schiex. 2006 and 2007 Max-SAT Evaluations: Contributed instances. *Journal on Satisfiability, Boolean Modeling and Computation*, 4(2-4):239–250, 2008.
- [HLH19] M. Hague, A. W. Lin, and C.-D. Hong. CSS minification via constraint solving. In *Transactions on Programming Languages and Systems*. ACM, 2019.
- [HLO08] F. Heras, J. Larrosa, and A. Oliveras. MiniMaxSAT: An efficient weighted Max-SAT solver. *Journal of Artificial Intelligence Research*, 31:1–32, 2008.
- [HMM15] F. Heras, A. Morgado, and J. Marques-Silva. MaxSAT-based encodings for group MaxSAT. *AI Communications*, 28(2):195–214, 2015.
- [HMS12] S. Hölldobler, N. Manthey, and P. Steinke. A compact encoding of pseudo-boolean constraints into SAT. In B. Glimm and A. Krüger, editors, *KI 2012: Advances in Artificial Intelligence - 35th Annual German Conference on AI, Saarbrücken, Germany, September 24-27, 2012. Proceedings*, volume 7526 of *Lecture Notes in Computer Science*, pages 107–118. Springer, 2012.
- [HPJ⁺17] A. Hyttinen, S. M. Plis, M. Järvisalo, F. Eberhardt, and D. Danks. A constraint optimization approach to causal discovery from subsampled time series data. *International Journal of Approximate Reasoning*, 90:208–225, 2017.
- [HSJ17] A. Hyttinen, P. Saikko, and M. Järvisalo. A core-guided approach to learning optimal causal graphs. In C. Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Aus-*

- tralia, August 19-25, 2017, pages 645–651. ijcai.org, 2017.
- [HST17] M. Heizmann, C. Schilling, and D. Tischner. Minimization of visibly pushdown automata using partial Max-SAT. In A. Legay and T. Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, volume 10206 of *Lecture Notes in Computer Science*, pages 461–478. Springer, 2017.
- [IJM14] A. Ignatiev, M. Janota, and J. Marques-Silva. Towards efficient optimization in package management systems. In P. Jalote, L. C. Briand, and A. van der Hoek, editors, *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014, Proceedings, Part I*, volume 10206 of *Lecture Notes in Computer Science*, pages 461–478. Springer, 2017.
- [IMM⁺14] A. Ignatiev, A. Morgado, V. M. Manquinho, I. Lynce, and J. Marques-Silva. Progression in maximum satisfiability. In T. Schaub, G. Friedrich, and B. O’Sullivan, editors, *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 453–458. IOS Press, 2014.
- [IMM17] A. Ignatiev, A. Morgado, and J. Marques-Silva. On Tackling the Limits of Resolution in SAT Solving. In S. Gaspers and T. Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10491 of *Lecture Notes in Computer Science*, pages 164–183. Springer, 2017.
- [IPLM15] A. Ignatiev, A. Previti, M. H. Liffiton, and J. Marques-Silva. Smallest MUS extraction with minimal hitting set dualization. In G. Pesant, editor, *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings*, pages 173–182, 2015.
- [IPNM18] A. Ignatiev, F. Pereira, N. Narodytska, and J. Marques-Silva. A SAT-based approach to learn explainable decision sets. In D. Galmiche, S. Schulz, and R. Sebastiani, editors, *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*, volume 10900 of *Lecture Notes in Computer Science*, pages 627–645. Springer, 2018.
- [JHB12] M. Järvisalo, M. Heule, and A. Biere. Inprocessing rules. In B. Gramlich, D. Miller, and U. Sattler, editors, *Auto-*

- mated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, volume 7364 of *Lecture Notes in Computer Science*, pages 355–370. Springer, 2012.
- [JKMR18] S. Joshi, P. Kumar, R. Martins, and S. Rao. Approximation strategies for incomplete MaxSAT. In J. N. Hooker, editor, *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, pages 219–228. Springer, 2018.
- [JM11] M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In M. W. Hall and D. A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 437–446. ACM, 2011.
- [JMM15] S. Joshi, R. Martins, and V. M. Manquinho. Generalized totalizer encoding for pseudo-boolean constraints. In G. Pesant, editor, *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings*, volume 9255 of *Lecture Notes in Computer Science*, pages 200–209. Springer, 2015.
- [Kar10] R. M. Karp. Implicit hitting set problems and multi-genome alignment. In A. Amir and L. Parida, editors, *Combinatorial Pattern Matching, 21st Annual Symposium, CPM 2010, New York, NY, USA, June 21-23, 2010. Proceedings*, volume 6129 of *Lecture Notes in Computer Science*, page 151. Springer, 2010.
- [KBSJ17] T. Korhonen, J. Berg, P. Saikko, and M. Järvisalo. MaxPre: An extended MaxSAT preprocessor. In S. Gaspers and T. Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10491 of *Lecture Notes in Computer Science*, pages 449–456. Springer, 2017.
- [Kle11] J. D. Kleer. Hitting set algorithms for model-based diagnosis. In *Proceedings of the 22nd International Workshop on Principles of Diagnosis (DX 2011)*, pages 100–105, 2011.
- [KMST10] S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney. ISAC - instance-specific algorithm configuration. In H. Coelho, R. Studer, and M. J. Wooldridge, editors, *ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 751–756. IOS Press, 2010.
- [KZFH12] M. Koshimura, T. Zhang, H. Fujita, and R. Hasegawa. QMaxSAT: A partial Max-SAT solver. *Journal of Satisfiability*

- ity, *Boolean Modeling and Computation*, 8(1/2):95–100, 2012.
- [LC18] Z. Lei and S. Cai. Solving (weighted) partial MaxSAT by dynamic local search for SAT. In J. Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden.*, pages 1346–1352. ijcai.org, 2018.
- [LCSH17] C. Luo, S. Cai, K. Su, and W. Huang. CCEHC: an efficient local search algorithm for weighted partial maximum satisfiability. *Artificial Intelligence*, 243:26–44, 2017.
- [LCW⁺15] C. Luo, S. Cai, W. Wu, Z. Jie, and K. Su. CCLS: an efficient local search algorithm for weighted maximum satisfiability. *IEEE Transactions on Computers*, 64(7):1830–1843, 2015.
- [LH05] J. Larrosa and F. Heras. Resolution in Max-SAT and its relation to local consistency in weighted CSPs. In L. P. Kaelbling and A. Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, pages 193–198. Professional Book Center, 2005.
- [LJX15] C. Li, H. Jiang, and R. Xu. Incremental MaxSAT reasoning to reduce branches in a branch-and-bound algorithm for Max-Clique. In C. Dhaenens, L. Jourdan, and M. Marmion, editors, *Learning and Intelligent Optimization - 9th International Conference, LION 9, Lille, France, January 12-15, 2015. Revised Selected Papers*, volume 8994 of *Lecture Notes in Computer Science*, pages 268–274. Springer, 2015.
- [LK12] P.-C. K. Lin and S. P. Khatri. Application of Max-SAT-based ATPG to optimal cancer therapy design. *BMC Genomics*, 13, 2012.
- [LM11] I. Lynce and J. Marques-Silva. Restoring CSP satisfiability with MaxSAT. *Fundamenta Informatica*, 107(2-3):249–266, 2011.
- [LMMP10] C. M. Li, F. Manyà, N. O. Mohamedou, and J. Planes. Resolution-based lower bounds in MaxSAT. *Constraints*, 15(4):456–484, 2010.
- [LQ10] C. M. Li and Z. Quan. An efficient branch-and-bound algorithm based on MaxSAT for the maximum clique problem. In M. Fox and D. Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press, 2010.
- [LZK16] X. Liao, H. Zhang, and M. Koshimura. Reconstructing AES key schedule images with SAT and MaxSAT. *IEICE Transactions*, 99-D(1):141–150, 2016.
- [LZMS12] C. M. Li, Z. Zhu, F. Manyà, and L. Simon. Optimizing with minimum satisfiability. *Artificial Intelligence*, 190:32–44, 2012.
- [MAGL11] J. Marques-Silva, J. Argelich, A. Graça, and I. Lynce. Boolean lexicographic optimization: algorithms & applications. *An-*

- nals of Mathematics and Artificial Intelligence*, 62(3-4):317–343, 2011.
- [Man12] N. Manthey. Coprocessor 2.0 - A flexible CNF simplifier - (tool presentation). In A. Cimatti and R. Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 436–441. Springer, 2012.
- [Mar17] R. Martins. Solving RNA alignment with MaxSAT. In *MaxSAT Evaluation 2017: Solver and Benchmark Descriptions*, volume B-2017-2 of *Department of Computer Science Series of Publications B*, pages 29–30. University of Helsinki, 2017.
- [MBM16] C. J. Muise, J. C. Beck, and S. A. McIlraith. Optimal partial-order plan relaxation via MaxSAT. *Journal of Artificial Intelligence Research*, 57:113–149, 2016.
- [MDM14] A. Morgado, C. Dodaro, and J. Marques-Silva. Core-guided MaxSAT with soft cardinality constraints. In B. O’Sullivan, editor, *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, volume 8656 of *Lecture Notes in Computer Science*, pages 564–573. Springer, 2014.
- [MHJ⁺13] J. Marques-Silva, F. Heras, M. Janota, A. Previt, and A. Belov. On computing minimal correction subsets. In F. Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 615–622. IJCAI/AAAI, 2013.
- [MHM12] A. Morgado, F. Heras, and J. Marques-Silva. Improvements to core-guided binary search for MaxSAT. In A. Cimatti and R. Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 284–297. Springer, 2012.
- [MIM17] J. Marques-Silva, A. Ignatiev, and A. Morgado. Horn maximum satisfiability: Reductions, algorithms and applications. In E. C. Oliveira, J. Gama, Z. A. Vale, and H. L. Cardoso, editors, *Progress in Artificial Intelligence - 18th EPIA Conference on Artificial Intelligence, EPIA 2017, Porto, Portugal, September 5-8, 2017, Proceedings*, volume 10423 of *Lecture Notes in Computer Science*, pages 681–694. Springer, 2017.
- [MJIM15] J. Marques-Silva, M. Janota, A. Ignatiev, and A. Morgado. Efficient model based diagnosis with maximum satisfiability. In Q. Yang and M. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 1966–1972. AAAI Press, 2015.
- [MJML14a] R. Martins, S. Joshi, V. Manquinho, and I. Lynce. On using in-

- cremental encodings in unsatisfiability-based MaxSAT solving. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:59–81, 2014.
- [MJML14b] R. Martins, S. Joshi, V. M. Manquinho, and I. Lynce. Incremental cardinality constraints for MaxSAT. In B. O’Sullivan, editor, *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, volume 8656 of *Lecture Notes in Computer Science*, pages 531–548. Springer, 2014.
- [MKTR16] T. Marcel Kevin and S. Tilak Raj. Finding pre-production vehicle configurations using a MaxSAT framework. In *Proceedings of the 18th International Configuration Workshop*, pages 117–122. École des Mines d’Albi-Carmaux, 2016.
- [MLM13] A. Morgado, M. H. Liffiton, and J. Marques-Silva. MaxSAT-based MCS enumeration. In A. Biere, A. Nahir, and T. E. J. Vos, editors, *Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers*, volume 7857 of *Lecture Notes in Computer Science*, pages 86–101. Springer, 2013.
- [MM08] J. Marques-Silva and V. M. Manquinho. Towards more effective unsatisfiability-based maximum satisfiability algorithms. In H. K. Büning and X. Zhao, editors, *Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings*, volume 4996 of *Lecture Notes in Computer Science*, pages 225–230. Springer, 2008.
- [MM10] A. Morgado and J. Marques-Silva. Combinatorial optimization solutions for the maximum quartet consistency problem. *Fundamenta Informatica*, 102(3-4):363–389, 2010.
- [MM11] A. Morgado and J. Marques-Silva. On validating boolean optimizers. In *IEEE 23rd International Conference on Tools with Artificial Intelligence, ICTAI 2011, Boca Raton, FL, USA, November 7-9, 2011*, pages 924–926, 2011.
- [MM18] D. Maliotov and K. S. Meel. MLIC: A MaxSAT-based framework for learning interpretable classification rules. In J. N. Hooker, editor, *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, pages 312–327. Springer, 2018.
- [MML11] R. Martins, V. M. Manquinho, and I. Lynce. Exploiting cardinality encodings in parallel maximum satisfiability. In *IEEE 23rd International Conference on Tools with Artificial Intelligence, ICTAI 2011, Boca Raton, FL, USA, November 7-9, 2011*, pages 313–320. IEEE Computer Society, 2011.
- [MML12] R. Martins, V. M. Manquinho, and I. Lynce. Parallel search

- for maximum satisfiability. *AI Communications*, 25(2):75–95, 2012.
- [MML13] R. Martins, V. M. Manquinho, and I. Lynce. Community-based partitioning for MaxSAT solving. In M. Jarvisalo and A. V. Gelder, editors, *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, volume 7962 of *Lecture Notes in Computer Science*, pages 182–191. Springer, 2013.
- [MML14] R. Martins, V. M. Manquinho, and I. Lynce. Open-WBO: A modular MaxSAT solver. In C. Sinz and U. Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 438–445. Springer, 2014.
- [MMP09] V. M. Manquinho, J. Marques-Silva, and J. Planes. Algorithms for weighted boolean optimization. In O. Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 495–508. Springer, 2009.
- [MNRS17] F. Manyà, S. Negrete, C. Roig, and J. R. Soler. A MaxSAT-based approach to the team composition problem in a classroom. In G. Sukthankar and J. A. Rodríguez-Aguilar, editors, *Autonomous Agents and Multiagent Systems - AAMAS 2017 Workshops, Visionary Papers, São Paulo, Brazil, May 8-12, 2017, Revised Selected Papers*, volume 10643 of *Lecture Notes in Computer Science*, pages 164–173. Springer, 2017.
- [MP07] J. Marques-Silva and J. Planes. On using unsatisfiability for solving maximum satisfiability. *CoRR*, abs/0712.1097, 2007.
- [MPLM08] P. J. Matos, J. Planes, F. Letombe, and J. Marques-Silva. A MAX-SAT algorithm portfolio. In M. Ghallab, C. D. Spyropoulos, N. Fakotakis, and N. M. Avouris, editors, *ECAI 2008 - 18th European Conference on Artificial Intelligence, Patras, Greece, July 21-25, 2008, Proceedings*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 911–912. IOS Press, 2008.
- [MPM15] C. Mencía, A. Previti, and J. Marques-Silva. Literal-based MCS extraction. In Q. Yang and M. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 1973–1979. AAAI Press, 2015.
- [MPS14] N. Manthey, T. Philipp, and P. Steinke. A more compact translation of pseudo-boolean constraints into CNF such that generalized arc consistency is maintained. In C. Lutz and M. Thielscher, editors, *KI 2014: Advances in Artificial In-*

- telligence - 37th Annual German Conference on AI, Stuttgart, Germany, September 22-26, 2014. *Proceedings*, volume 8736 of *Lecture Notes in Computer Science*, pages 123–134. Springer, 2014.
- [MS17] R. Martins and J. Sherry. Lisbon wedding: Seating arrangements using MaxSAT. In *MaxSAT Evaluation 2017: Solver and Benchmark Descriptions*, volume B-2017-2 of *Department of Computer Science Series of Publications B*, pages 25–26. University of Helsinki, 2017.
- [MZNN15a] R. Mangal, X. Zhang, A. V. Nori, and M. Naik. A user-guided approach to program analysis. In E. D. Nitto, M. Harman, and P. Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 462–473. ACM, 2015.
- [MZNN15b] R. Mangal, X. Zhang, A. V. Nori, and M. Naik. Volt: A lazy grounding framework for solving very large MaxSAT instances. In M. Heule and S. Weaver, editors, *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, volume 9340 of *Lecture Notes in Computer Science*, pages 299–306. Springer, 2015.
- [Nad18] A. Nadel. Solving MaxSAT with bit-vector optimization. In O. Beyersdorff and C. M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10929 of *Lecture Notes in Computer Science*, pages 54–72. Springer, 2018.
- [NB14] N. Narodytska and F. Bacchus. Maximum satisfiability using core-guided MaxSAT resolution. In C. E. Brodley and P. Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pages 2717–2723. AAAI Press, 2014.
- [NLH⁺04] E. Nudelman, K. Leyton-Brown, H. H. Hoos, A. Devkar, and Y. Shoham. Understanding random SAT: beyond the clauses-to-variables ratio. In M. Wallace, editor, *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, volume 3258 of *Lecture Notes in Computer Science*, pages 438–452. Springer, 2004.
- [NMJ⁺15] M. Neves, R. Martins, M. Janota, I. Lynce, and V. M. Manquinho. Exploiting resolution-based representations for MaxSAT solving. In M. Heule and S. Weaver, editors, *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, volume 9340 of *Lecture Notes in Computer*

- Science*, pages 272–286. Springer, 2015.
- [NW99] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. Wiley-Interscience, 1999.
- [NWJ16a] A. Niskanen, J. P. Wallner, and M. Järvisalo. Optimal status enforcement in abstract argumentation. In S. Kambhampati, editor, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 1216–1222. IJCAI/AAAI Press, 2016.
- [NWJ16b] A. Niskanen, J. P. Wallner, and M. Järvisalo. Synthesizing argumentation frameworks from examples. In G. A. Kaminka, M. Fox, P. Bouquet, E. Hüllermeier, V. Dignum, F. Dignum, and F. van Harmelen, editors, *ECAI 2016 - 22nd European Conference on Artificial Intelligence, 29 August-2 September 2016, The Hague, The Netherlands - Including Prestigious Applications of Artificial Intelligence (PAIS 2016)*, volume 285 of *Frontiers in Artificial Intelligence and Applications*, pages 551–559. IOS Press, 2016.
- [OLH⁺13] T. Ogawa, Y. Liu, R. Hasegawa, M. Koshimura, and H. Fujita. Modulo based CNF encoding of cardinality constraints and its application to MaxSAT solvers. In *25th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2013, Herndon, VA, USA, November 4-6, 2013*, pages 9–17. IEEE Computer Society, 2013.
- [Pap94] C. H. Papadimitriou. *Computational Complexity*. Addison Wesley Longman, 1994.
- [Par02] J. D. Park. Using weighted MAX-SAT engines to solve MPE. In R. Dechter and R. S. Sutton, editors, *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada.*, pages 682–687. AAAI Press / The MIT Press, 2002.
- [PFL14] P. Parviainen, H. S. Farahani, and J. Lagergren. Learning bounded tree-width Bayesian networks using integer linear programming. In J. Corander and S. Kaski, editors, *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics, AISTATS 2014, Reykjavik, Iceland, April 22-25, 2014*, volume 33 of *JMLR Workshop and Conference Proceedings*, pages 751–759. JMLR.org, 2014.
- [PRB18] T. Paxian, S. Reimer, and B. Becker. Dynamic polynomial watchdog encoding for solving weighted MaxSAT. In O. Beyersdorff and C. M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10929 of *Lecture Notes in Computer Science*, pages 37–53. Springer, 2018.

- [Rei87] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.
- [SA18] A. Shabani and B. Alizadeh. PMTP: A MAX-SAT based approach to detect hardware trojan using propagation of maximum transition probability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.
- [Sai15] P. Saikko. Re-implementing and extending a hybrid SAT-IP approach to maximum satisfiability. Master’s thesis, University of Helsinki, 2015.
- [SBJ16] P. Saikko, J. Berg, and M. Järvisalo. LMHS: A SAT-IP hybrid MaxSAT solver. In N. Creignou and D. L. Berre, editors, *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 539–546. Springer, 2016.
- [Sin05] C. Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In P. van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, volume 3709 of *Lecture Notes in Computer Science*, pages 827–831. Springer, 2005.
- [SMJ15] P. Saikko, B. Malone, and M. Järvisalo. MaxSAT-based cutting planes for learning graphical models. In L. Michel, editor, *Integration of AI and OR Techniques in Constraint Programming - 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings*, volume 9075 of *Lecture Notes in Computer Science*, pages 347–356. Springer, 2015.
- [SMV⁺07] S. Safarpour, H. Mangassarian, A. G. Veneris, M. H. Liffiton, and K. A. Sakallah. Improved design debugging using maximum satisfiability. In *Formal Methods in Computer-Aided Design, 7th International Conference, FMCAD 2007, Austin, Texas, USA, November 11-14, 2007, Proceedings*, pages 13–19. IEEE Computer Society, 2007.
- [SZGN17] X. Si, X. Zhang, R. Grigore, and M. Naik. Maximum satisfiability in software analysis: Applications and techniques. In R. Majumdar and V. Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10427 of *Lecture Notes in Computer Science*, pages 68–94. Springer, 2017.
- [TDFDGDSJ⁺17] A. B. Trindade, R. De Faria Degelo, E. Galvão Dos Santos Junior, H. I. Ismail, H. Cruz Da Silva, and L. C. Cordeiro. Multi-core model checking and maximum satisfiability applied to hardware-software partitioning. *International Journal of Engineering Science*, 9(6):570–582, 2017.
- [TLM16] M. Terra-Neves, I. Lynce, and V. M. Manquinho. Non-

- portfolio approaches for distributed maximum satisfiability. In *28th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2016, San Jose, CA, USA, November 6-8, 2016*, pages 436–443. IEEE Computer Society, 2016.
- [vdTHB12] P. van der Tak, M. Heule, and A. Biere. Concurrent cube-and-conquer - (poster presentation). In A. Cimatti and R. Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 475–476. Springer, 2012.
- [War98] J. P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, 68(2):63–69, 1998.
- [WJH18] A. A. Wakrime, S. Jabbour, and N. Hameurlain. A MaxSAT based approach for QoS cloud services. *International Journal of Parallel, Emergent and Distributed Systems*, 2018.
- [WNJ17] J. P. Wallner, A. Niskanen, and M. Järvisalo. Complexity results and algorithms for extension enforcement in abstract argumentation. *Journal of Artificial Intelligence Research*, 60:1–40, 2017.
- [Wol98] L. A. Wolsey. *Integer Programming*. John Wiley, 1998.
- [WQL09] G. T. Wickramaarachchi, W. H. Qardaji, and N. Li. An efficient framework for user authorization queries in RBAC systems. In B. Carminati and J. Joshi, editors, *14th ACM Symposium on Access Control Models and Technologies, SACMAT 2009, Stresa, Italy, June 3-5, 2009, Proceedings*, pages 23–32. ACM, 2009.
- [XHHL08] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Satzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008.
- [XRS03] H. Xu, R. A. Rutenbar, and K. Sakallah. sub-SAT: a formulation for relaxed Boolean satisfiability with applications in routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(6):814–820, 2003.
- [ZB12] L. Zhang and F. Bacchus. MAXSAT heuristics for cost optimal planning. In J. Hoffmann and B. Selman, editors, *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada*. AAAI Press, 2012.
- [ZLMA12] Z. Zhu, C. M. Li, F. Manyà, and J. Argelich. A new encoding from MinSAT into MaxSAT. In M. Milano, editor, *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, volume 7514 of *Lecture Notes in Computer Science*, pages 455–463. Springer, 2012.
- [ZMNN16] X. Zhang, R. Mangal, A. V. Nori, and M. Naik. Query-guided maximum satisfiability. In R. Bodík and R. Majumdar, edi-

tors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 109–122. ACM, 2016.

- [ZWM11] C. S. Zhu, G. Weissenbacher, and S. Malik. Post-silicon fault localisation using maximum satisfiability and backbones. In P. Bjesse and A. Slobodová, editors, *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, pages 63–66. FMCAD Inc., 2011.

This page intentionally left blank

Chapter 25

Model Counting

Carla P. Gomes, Ashish Sabharwal, and Bart Selman

Propositional model counting or #SAT is the problem of computing the number of models for a given propositional formula, i.e., the number of distinct truth assignments to variables for which the formula evaluates to TRUE. For a propositional formula F , we will use $\#F$ to denote the model count of F . This problem is also referred to as the solution counting problem for SAT. It generalizes SAT and is the canonical #P-complete problem. There has been significant theoretical work trying to characterize the worst-case complexity of counting problems, with some surprising results such as model counting being hard even for some polynomial-time solvable problems like 2-SAT.

The model counting problem presents fascinating challenges for practitioners and poses several new research questions. Efficient algorithms for this problem will have a significant impact on many application areas that are inherently beyond SAT ('beyond' under standard complexity theoretic assumptions), such as bounded-length adversarial and contingency planning, and probabilistic reasoning. For example, various probabilistic inference problems, such as Bayesian net reasoning, can be effectively translated into model counting problems [cf. BDP03, Dar05, LMP01, Par02, Rot96, SBK05b]. Another application is in the study of hard combinatorial problems, such as combinatorial designs, where the number of solutions provides further insights into the problem. Even finding a single solution can be a challenge for such problems; counting the number of solutions is much harder. Not surprisingly, the largest formulas we can solve for the model counting problem with state-of-the-art model counters are orders of magnitude smaller than the formulas we can solve with the best SAT solvers. Generally speaking, current exact counting methods can tackle problems with a couple of hundred variables, while approximate counting methods push this to around 1,000 variables.

#SAT can be solved, in principle and to an extent in practice, by extending the two most successful frameworks for SAT algorithms, namely, DPLL and local search. However, there are some interesting issues and choices that arise when extending SAT-based techniques to this harder problem. In general, solving #SAT requires the solver to, in a sense, be cognizant of all solutions in the search space, thereby reducing the effectiveness and relevance of commonly used SAT heuristics designed to quickly narrow down the search to a single solution. The resulting

scalability challenge has drawn many satisfiability researchers to this problem, and to the related problem of sampling solutions uniformly at random.

We will divide practical model counting techniques we consider into two main categories: *exact counting* and *approximate counting*, discussed in Sections 25.2 and 25.3, respectively. Within exact counting, we will distinguish between methods based on *DPLL-style* exhaustive search (Section 25.2.1) and those based on “knowledge compilation” or conversion of the formula into certain *normal forms* (Section 25.2.2). Within approximate counting, we will distinguish between methods that provide fast *estimates without any guarantees* (Section 25.3.1) and methods that provide lower or upper *bounds with a correctness guarantee*, often in a probabilistic sense and also in a statistical sense (Section 25.3.2).

We would like to note that there are several other directions of research related to model counting that we will not cover here. For example, Nishimura et al. [NRS06] explore the concept of “backdoors” for #SAT, and show how the vertex cover problem can be used to identify small such backdoors based on so-called cluster formulas. Bacchus et al. [BDP03] consider structural restrictions on the formula and propose an algorithm for #SAT whose complexity is polynomial in n (the number of variables) and exponential in the “branch-width” of the underlying constraint graph. Gottlob et al. [GSS02] provide a similar result in terms of “tree-width”. Fischer et al. [FMR08] extend this to a similar result in terms of “cluster-width” (which is never more than tree-width, and sometimes smaller). There is also complexity theoretic work on this problem by the theoretical computer science community. While we do provide a flavor of this work (Section 25.1), our focus will mostly be on techniques that are available in the form of implemented and tested model counters.

25.1. Computational Complexity of Model Counting

We begin with a relatively brief discussion of the theoretical foundations of the model counting problem. The reader is referred to standard complexity texts [cf. Pap94] for a more detailed treatment of the subject.

Given any problem in the class NP, say SAT or CLIQUE, one can formulate the corresponding *counting problem*, asking *how many solutions exist* for a given instance? More formally, given a polynomial-time decidable relation Q ,¹ the corresponding counting problem asks: given x as input, how many y ’s are there such that $(x, y) \in Q$? For example, if Q is the relation “ y is a truth assignment that satisfies the propositional expression x ” then the counting problem for Q is the propositional model counting problem, #SAT. Similarly, if Q is the relation “ y is a clique in the graph x ” then the counting problem for Q is #CLIQUE. The complexity class #P (pronounced “number P” or “sharp P”) consists of all counting problems associated with such polynomial-time decidable relations. Note that the corresponding problem in NP asks: given x , *does there exist a y such that $(x, y) \in Q$?*

The notion of *completeness* for #P is defined essentially in the usual way, with a slight difference in the kind of reduction used. A problem A is #P-complete if

¹Technically, Q must also be polynomially balanced, that is, for each x , the only possible y ’s with $(x, y) \in Q$ satisfy $|y| \leq |x|^k$ for a constant k .

(1) A is in $\#P$, and (2) for every problem B in $\#P$, there exists a polynomial-time *counting reduction* from B to A . A counting reduction is an extension of the reductions one often encounters between NP-complete decision problems, and applies to function computation problems. There are two parts to a counting reduction from B to A : a polynomial-time computable function R that maps an instance z of B to an instance $R(z)$ of A , and a polynomial-time computable function S that recovers from the count N of $R(z)$ the count $S(N)$ of z . In effect, given an algorithm for the counting problem A , the relations R and S together give us a recipe for converting that algorithm into one for the counting problem B with only a polynomial overhead.

Conveniently, many of the known reductions between NP-complete problems are already *parsimonious*, that is, they preserve the number of solutions during the translation. Therefore, these reductions can be directly taken to be the R part of a counting reduction, with the trivial identity function serving as the S part, thus providing an easy path to proving $\#P$ -completeness. In fact, one can construct a parsimonious version of the Cook-Levin construction, thereby showing that $\#SAT$ is a canonical $\#P$ -complete problem. As it turns out, the solution counting variants of all six basic NP-complete problems listed by Garey and Johnson [GJ79], and of many more NP-complete problems, are known to be $\#P$ -complete.²

In his seminal paper, Valiant [Val79] proved that, quite surprisingly, the solution counting variants of polynomial-time solvable problems can also be $\#P$ -complete. Such problems, in the class P , include 2-SAT, Horn-SAT, DNF-SAT, bipartite matching, etc. What Valiant showed is that the problem PERM of computing the *permanent* of a 0-1 matrix, which is equivalent to counting the number of perfect matchings in a bipartite graph or $\#BIP$ -MATCHING, is $\#P$ -complete. On the other hand, the corresponding search problem of computing a *single* perfect matching in a bipartite graph can be solved in deterministic polynomial time using, e.g., a network flow algorithm. Therefore, unless $P=NP$, there does not exist a *parsimonious* reduction from SAT to BIP-MATCHING; such a reduction would allow one to solve any SAT instance in polynomial time by translating it to a BIP-MATCHING instance and checking for the existence of a perfect matching in the corresponding bipartite graph. Valiant instead argued that there is a smart way to *indirectly* recover the answer to a $\#SAT$ instance from the answer to the corresponding PERM (or $\#BIP$ -MATCHING) instance, using a non-identity polynomial-time function S in the above notation of counting reductions.

Putting counting problems in the traditional complexity hierarchy of decision problems, Toda [Tod89] showed that $\#SAT$ being $\#P$ -complete implies that it is no easier than solving a quantified Boolean formula (QBF) with a constant number (independent of n , the number of variables) of “there exist” and “forall” quantification levels in its variables. For a discussion of the QBF problem, see Part 2, Chapters 29-31 of this Handbook. Formally, Toda considered the decision problem class $P^{\#P}$ consisting of polynomial-time decision computations with “free” access to $\#P$ queries, and compared this with k -QBF, the subset of QBF instances that have exactly k quantifier alternations, and the infinite polynomial

²Note that a problem being NP-complete does not automatically make its solution counting variant $\#P$ -complete; one must demonstrate a polynomial time *counting* reduction.

hierarchy $\text{PH} = \bigcup_{k=1}^{\infty} k\text{-QBF}$.³ Combining his result with the relatively easy fact that counting problems can be solved in polynomial space, we have $\#\text{P}$ placed as follows in the worst-case complexity hierarchy:

$$\text{P} \subseteq \text{NP} \subseteq \text{PH} \subseteq \text{P}\#\text{P} \subseteq \text{PSPACE}$$

where PSPACE is the class of problems solvable in polynomial space, with QBF being the canonical PSPACE -complete problem. As a comparison, notice that SAT can be thought of as a QBF with exactly one level of “there exist” quantification for all its variables, and is thus a subset of 1-QBF . While the best known deterministic algorithms for SAT , $\#\text{SAT}$, and QBF all run in worst-case exponential time, it is widely believed—by theoreticians and practitioners alike—that $\#\text{SAT}$ and QBF are significantly harder to solve than SAT .

While $\#\text{P}$ is a class of function problems (rather than decision problems, for which the usual complexity classes like P and NP are defined), there does exist a natural variant of it called PP (for “probabilistic polynomial time”) which is a class of essentially equally hard decision problems. For a polynomial-time decidable relation Q , the corresponding PP problem asks: given x , is $(x, y) \in Q$ for *more than half* the y ’s? The class PP is known to contain both NP and co-NP , and is quite powerful. The proof of Toda’s theorem mentioned earlier in fact relies on the equality $\text{P}^{\text{PP}} = \text{P}\#\text{P}$ observed by Angluin [Ang80]. One can clearly answer the PP query for a problem given the answer to the corresponding $\#\text{P}$ query. The other direction is a little less obvious, and uses the fact that PP has the power to provide the “most significant bit” of the answer to the $\#\text{P}$ query and it is possible to obtain all bits of this answer by repeatedly querying the PP oracle on appropriately modified problem instances.

We close this section with a note that Karp and Luby [KL85] gave a Markov Chain Monte Carlo (MCMC) search based fully polynomial-time randomized approximation scheme (FPRAS) for the DNF-SAT model counting problem, and Karp et al. [KLM89] later improved its running time to yield the following: for any $\epsilon, \delta \in (0, 1)$, there exists a randomized algorithm that given F computes an ϵ -approximation to $\#\text{F}$ with correctness probability $1 - \delta$ in time $O(|F| \cdot 1/\epsilon^2 \cdot \ln(1/\delta))$, where $|F|$ denotes the size of F .

25.2. Exact Model Counting

We now move on to a discussion of some of the practical implementations of exact model counting methods which, upon termination, output the true model count of the input formula. The “model counters” we consider are CDP by Birnbaum and Lozinskii [BL99], ReIsat by Bayardo Jr. and Pehoushek [BP00], Cachet by Sang et al. [SBB⁺04], sharpSAT by Thurley [Thu06], and c2d by Darwiche [Dar04].

³For simplicity, we use $k\text{-QBF}$ to represent not only a subset of QBF formulas but also the complexity class of alternating Turing machines with k alternations, for which solving these formulas forms the canonical complete problem.

25.2.1. DPLL-Based Model Counters

Not surprisingly, the earliest practical approach for counting models is based on an extension of systematic DPLL-style SAT solvers. The idea, formalized early by Birnbaum and Lozinskii [BL99] in their model counter CDP, is to directly explore the complete search tree for an n -variable formula as in the usual DPLL search, pruning unsatisfiable branches based on falsified clauses and declaring a branch to be satisfied when all clauses have at least one TRUE literal. However, unlike the usual DPLL, when a branch is declared satisfied and the partial truth assignment at that point has t fixed variables (fixed either through the branching heuristic or by unit propagation), we associate 2^{n-t} solutions with this branch corresponding to the partial assignment being extended by all possible settings of the $n - t$ yet unset variables, backtrack to the last decision variable that can be flipped, flip that variable, and continue exploring the remaining search space. The model count for the formula is finally computed as the sum of such 2^{n-t} counts obtained over all satisfied branches. Although all practical implementations of DPLL have an iterative form, it is illustrative to consider CDP in a recursive manner, written here as Algorithm 25.1, where $\#F$ is computed as the sum of $\#F|_x$ and $\#F|_{\neg x}$ for a branch variable x , with the discussion above reflected in the two base cases of this recursion.

Algorithm 25.1: CDP ($F, 0$)

Input : A CNF formula F over n variables; a parameter t initially set to 0
Output : $\#F$, the model count of F
begin
 UnitPropagate(F)
 if F has an empty clause **then return** 0
 if all clauses of F are satisfied **then return** 2^{n-t}
 $x \leftarrow$ SelectBranchVariable(F)
 return CDP($F|_x, t + 1$) + CDP($F|_{\neg x}, t + 1$)
end

An interesting technical detail is that many of the modern implementations of DPLL do *not* maintain data structures that would allow one to easily check whether or not all clauses have been satisfied by the current partial assignment. In general, DPLL-style SAT solvers often do not explicitly keep track of the number of unsatisfied clauses. They only keep track of the number of assigned variables, and declare success when all variables have been assigned values and no clause is violated. Keeping track of the number of unsatisfied clauses is considered unnecessary because once a partial assignment happens to satisfy all clauses, further branching immediately sets all remaining variables to arbitrary values and obtains a complete satisfying assignment; complete satisfying assignments are indeed what many applications of SAT seek. DPLL-based model counters, on the other hand, do maintain this added information about how many clauses are currently satisfied and infer the corresponding 2^{n-t} counts. Having to enumerate each of the 2^{n-t} complete satisfying assignments instead would make the technique impractical.

Obtaining partial counts: As discussed above, a basic DPLL-based model counter works by using appropriate multiplication factors and continuing the search after a single solution is found. An advantage of this approach is that the model count is computed in an incremental fashion: if the algorithm runs out of a pre-specified time limit, it can still output a correct *lower bound* on the true model count, based on the part of the search space it has already explored. This can be useful in many applications, and has been the motivation for some new randomized techniques that provide fast lower bounds with probabilistic correctness guarantees (to be discussed in Section 25.3.2). In fact, a DPLL-based model counter can, in principle, also output a correct *upper bound* at any time: 2^n minus the sum of the 2^{n-t} style counts of un-satisfying assignments contained in all unsatisfiable branches explored till that point. Unfortunately, this is often not very useful in practice because the number of solutions of problems of interest is typically much smaller than the size of the search space. As a simple example, in a formula with 1000 variables and 2^{200} solutions, after having explored, say, a $1/16$ fraction of the search space, it is reasonable to expect the model counter to have seen roughly $2^{200}/16 = 2^{196}$ solutions (which would be a fairly good lower bound on the model count) while one would expect to have seen roughly $(2^{1000} - 2^{200})/16$ un-satisfying assignments (yielding a poor naïve upper bound of $2^{1000} - (2^{1000} - 2^{200})/16$, which is at least as large as 2^{999}). We will discuss a more promising, statistical upper bounding technique towards the end of this chapter.

Component analysis: Consider the constraint graph G of a CNF formula F . The vertices of G are the variables of F and there is an edge between two vertices if the corresponding variables appear together in some clause of F . Suppose G can be partitioned into disjoint components G_1, G_2, \dots, G_k where there is no edge connecting a vertex in one component to a vertex in another component, i.e., the variables of F corresponding to vertices in two different components do not appear together in any clause. Let F_1, F_2, \dots, F_k be the sub-formulas of F corresponding to the k components of G and restricted to the variables that appear in the corresponding component. Since the components are disjoint, it follows that every clause of F appears in a unique component, the sub-problems captured by the components are independent, and, most pertinent to this chapter, that $\#F = \#F_1 \times \#F_2 \times \dots \times \#F_k$. Thus, $\#F$ can be evaluated by identifying disjoint components of F , computing the model count of each component, and multiplying the results together.

This idea is implemented in one of the first effective exact model counters for SAT, called `ReIsat` [BP00], which extends a previously introduced DPLL-based SAT solver by the same name [BS97]. Components are identified dynamically as the underlying DPLL procedure attempts to extend a partial assignment. With each new extension, several clauses may be satisfied so that the constraint graph simplifies dynamically depending on the actual value assignment to variables. While such dynamic detection and exploitation of components has often been observed to be too costly for pure satisfiability testing,⁴ it certainly pays off well for the harder task of model counting. Note that for the correctness of the

⁴Only recently have SAT solvers begun to efficiently exploit partial component caching schemes [PD07].

method, all we need is that the components are disjoint. However, the components detected by `ReIsat` are, in fact, the *connected* components of the constraint graph of F , indicating that the full power of this technique is being utilized. One of the heuristic *optimizations* used in `ReIsat` is to attempt the most constrained sub-problems first. Another trick is to first check the satisfiability of every component, before attempting to count any. `ReIsat` also solves sub-problems in an interleaved fashion, dynamically jumping to another sub-problem if the current one turns out to be less constrained than initially estimated, resulting in a best-first search of the developing component tree. Finally, the component structure of the formula is determined lazily while backtracking, instead of eagerly before branch selection. This does not affect the search space explored but often reduces the component detection overhead for unsatisfiable branches.

Bayardo Jr. and Schrag [BS97] demonstrated through `ReIsat` that applying these ideas significantly improves performance over basic CDP, obtaining exact counts for several problems from graph coloring, planning, and circuit synthesis/analysis domains that could not be counted earlier. They also observed, quite interestingly, that the *peak of hardness* of model counting for random 3-SAT instances occurs at a very different clause-to-variable ratio than the peak of hardness of solving such formulas. These instances were found to be the hardest for model counting at a ratio of $\alpha \approx 1.5$, compared to $\alpha \approx 4.26$ which marks the peak of hardness for SAT solvers as well as the (empirical) point of phase transition in such formulas from being mostly satisfiable to mostly unsatisfiable. Bailey et al. [BDK07] followed up on this observation and provided further analytical and empirical results on the hardness peak and the corresponding phase transition of the decision variants of random counting problems.

Caching: As one descends the search tree of a DPLL-based model counter, setting variables and simplifying the formula, one may encounter sub-formulas that have appeared in an earlier branch of the search tree. If this happens, it would clearly be beneficial to be able to efficiently recognize this fact, and instead of re-computing the model count of the sub-formula from scratch, somehow “remember” the count computed for it earlier. This is, in principle, similar to the clause learning techniques used commonly in today’s SAT solvers, except that for the purposes of model counting, it is no longer possible to succinctly express the key knowledge learned from each previous sub-formula as a single “conflict clause” that, for SAT, quite effectively captures the “reason” for that sub-formula being unsatisfiable. For model counting, one must also store, in some form, a signature of the full satisfiable sub-formulas encountered earlier, along with their computed model counts. This is the essence of *formula caching* systems [BDP03, BIPS03, ML98]. While formula caching is theoretically appealing even for SAT, being more powerful than clause learning [BIPS03], its overhead is much more likely to be offset when applied to harder problems like #SAT.

Bacchus et al. [BDP03] considered three variants of caching schemes: simple caching (a.k.a. formula caching), component caching, and linear-space caching. They showed that of these, *component caching* holds the greatest promise, being theoretically competitive with (and sometimes substantially better than) some of the best known methods for Bayesian inference. Putting these ideas into prac-

tice, Sang et al. [SBB⁺04] created the model counter **Cachet**, which ingeniously combined component caching with traditional clause learning within the setup of model counting.⁵ **Cachet** is built upon the well-known SAT solver **zChaff** [MMZ⁺01]. It turns out that combining component caching and clause learning in a naïve way leads to subtle issues that would normally permit one to only compute a lower bound on the model count. This problem is taken care of in **Cachet** using so-called *sibling pruning*, which prevents the undesirable interaction between cached components and clause learning from spreading.

Taking these ideas further, Sang et al. [SBK05a] considered the efficiency of various heuristics used in SAT solvers, but now in the context of model counting with component caching. They looked at component selection strategies, variable selection branching heuristics, randomization, backtracking schemes, and cross-component implications. In particular, they showed that model counting works better with a variant of the conflict graph based branching heuristic employed by **zChaff**, namely VSIDS (variable state independent decaying sum). This variant is termed VSADS, for variable state *aware* decaying sum, which linearly interpolates between the original VSIDS score and a more traditional formula-dependent score based on the number of occurrences of each variable.

Improved caching and more reasoning at each node: An important concern in implementing formula caching or component caching in practice is the space requirement. While these concerns are already present even for clause learning techniques employed routinely by SAT solvers and have led to the development of periodic clause deletion mechanisms, the problem is clearly more severe when complete sub-formulas are cached. **sharpSAT** [Thu06] uses several ideas that let components be stored more succinctly. For example, all clauses of any component stored by **sharpSAT** have at least two unassigned literals (unit propagation takes care of any active clauses with only one unassigned literal), and it does not explicitly store any binary clauses of the original formula in the component signature (binary clauses belonging to the component have both literals unassigned and can thus be easily reconstructed from the set of variables associated with the component). Further, it only stores (1) the indices of the variables in the component and (2) the indices of the original clauses that belong to that component, rather than storing full clauses or the learned conflict clauses. This can, in principle, prohibit some components from being identified as identical when they would be identified as identical by **Cachet**, which stores full clauses. Nonetheless, these techniques together are shown to reduce the storage requirement by an order of magnitude or more compared to **Cachet**, and to often increase efficiency.

sharpSAT also uses a “look ahead” technique known in the SAT community as the *failed literal* rule (the author refers to it as *implicit BCP*). Here every so often one identifies a set of candidate variables for each of which the failed literal test is applied: if setting x to TRUE makes the current formula unsatisfiable, then assert $x = \text{FALSE}$ and simplify; otherwise, if setting x to FALSE makes the current formula unsatisfiable, then assert $x = \text{TRUE}$ and simplify. The technique is shown to pay off well while model counting several difficult instances.

⁵Note that clause learning and decomposition into components were already implemented in the model counter **RelSAT**, but no caching.

Davies and Bacchus [DB07] have shown that employing more reasoning at each node of the DPLL search tree can significantly speed-up the model counting process.⁶ Specifically, they use hyper-binary resolution and equality reduction in addition to unit propagation, which simplifies the formula and often results in more efficient component detection and caching, and sometimes even stronger component division.

25.2.2. Counting Using Knowledge Compilation

A different approach for exact model counting is to convert or *compile* the given CNF formula into another logical form from which the count can be deduced easily, i.e., in time polynomial in the size of the formula in the new logical form. For example, in principle, one could convert the formula into a binary decision diagram or BDD [Bry86] and then “read off” the solution count by traversing the BDD from the leaf labeled “1” to the root. One advantage of this methodology is that once resources have been spent on compiling the formula into this new form, several complex queries can potentially be answered fairly quickly, often with a linear time traversal with simple book keeping. For instance, with BDDs, one can easily answer queries about satisfiability, being a tautology, logical equivalence to another formula, number of solutions, etc.

A knowledge compilation alternative was introduced by Darwiche [Dar04] in a compiler called *c2d*, which converts the given CNF formula into *deterministic, decomposable negation normal form* or d-DNNF. The DNNF form [Dar01, DM02] is a strict superset of ordered BDDs (in the sense that an ordered BDD can be converted in linear time into DNNF), and often more succinct. While a BDD is structurally quite different from a CNF style representation of a Boolean function,⁷ the negation normal form or NNF underlying d-DNNF is very much like CNF. Informally, one can think of a CNF formula as a 4-layer directed acyclic graph, with the root node labeled with AND or \wedge , pointing to all nodes in the next layer corresponding to clauses and labeled with OR or \vee , and each clause node pointing either directly or through a layer-3 node labeled NOT or \neg to nodes labeled with variable names, one for each variable; this represents the conjunction of disjunctions that defines CNF. In contrast, an NNF formula is defined by a rooted directed acyclic graph in which there is no restriction on the depth, each non-leaf node is labeled with either \wedge or \vee , each leaf node is labeled with either a variable or its negation, and the leaf nodes only have incoming edges (as before). Thus, unlike CNF, one may have several levels of alternations between \wedge and \vee nodes, but all negations are pushed down all the way to the leaf nodes. There are, in general, twice as many leaf nodes as variables.

In order to exploit NNF for model counting, one must add two features to it, decomposability and determinism:

- i. *Decomposability* says that for the children A_1, A_2, \dots, A_s of a node A^{AND} labeled \wedge , the variables appearing in each pair of the A_i 's must be disjoint, i.e., the logical expression at an AND-node can be decomposed into disjoint

⁶In SAT solving, this extra reasoning was earlier observed to not be cost effective.

⁷A BDD is more akin to the search space of a DPLL-style process, with nodes corresponding to branching on a variable by fixing it to TRUE or FALSE.

components corresponding to its children. For model counting, this translates into $\#f^{\text{AND}} = \#f_1 \times \#f_2 \times \dots \times \#f_s$, where f^{AND} and f_i denote the Boolean functions captured by A^{AND} and A_i , respectively.

- ii. In a similar manner, *determinism* says that the children B_1, B_2, \dots, B_t of a node B^{OR} labeled \vee do not have any common solutions, i.e., the logical conjunction of the Boolean functions represented by any two children of an OR-node is inconsistent. For model counting, this translates into $\#f^{\text{OR}} = \#f_1 + \#f_2 + \dots + \#f_s$, where f^{OR} and f_i denote the Boolean functions captured by B^{OR} and B_i , respectively.

The above properties suggest a simple model counting algorithm which computes $\#F$ from a d-DNNF representation of F , by performing a topological traversal of the underlying acyclic graph starting from the leaf nodes. Specifically, each leaf node is assigned a count of 1, the count of each \wedge node is computed as the product of the counts of its children, and the count of each \vee node is computed as the sum of the counts of its children. The count associated with the root of the graph is reported as the model count of F .

In the simplified d-DNNF form generated by `c2d`, every \vee node has exactly two children, and the node has as its secondary label the identifier of a variable that is guaranteed to appear as `TRUE` in all solutions captured by one child and as `FALSE` in all solutions captured by the other child. `c2d` can also be asked to compile the formula into a *smoothed* form, where each child of an \vee node has the same number of variables.

Inside `c2d`, the compilation of the given CNF formula F into d-DNNF is done by first constructing a decomposition tree or *dtree* for F , which is a binary tree whose leaves are tagged with the clauses of F and each of whose non-leaf vertices has a set of variables, called the *separator*, associated with it. The separator is simply the set of variables that are shared by the left and right branches of the node, the motivation being that once these variables have been assigned truth values, the two resulting sub-trees will have disjoint sets of variables. `c2d` uses an exhaustive version of the DPLL procedure to construct the dtree and compile it to d-DNNF, by ensuring that the separator variables for each node are either instantiated to various possible values (and combined using \vee nodes) or no longer shared between the two subtrees (perhaps because of variable instantiations higher up in the dtree or from the resulting unit-propagation simplifications). Once the separator variables are instantiated, the resulting components become disjoint and are therefore combined using \wedge nodes.

`c2d` has been demonstrated to be quite competitive on several classes of formulas, and sometimes more efficient than DPLL-based exact counters like `Cachet` and `ReIsat` even for obtaining a single overall model count. For applications that make several counting-related queries on a single formula (such as “marginal probability” computation or identifying “backbone variables” in the solution set), this knowledge compilation approach has a clear “re-use” advantage over traditional DPLL-based counters. This approach is currently being explored also for computing connected clusters in the solution space of a formula.

25.3. Approximate Model Counting

Most exact counting methods, especially those based on DPLL search, essentially attack a #P-complete problem head on—by exhaustively exploring the raw combinatorial search space. Consequently, these algorithms often have difficulty scaling up to larger problem sizes. For example, perhaps it is too much to expect a fast algorithm to be able to precisely distinguish between a formula having 10^{70} and $10^{70} + 1$ solutions. Many applications of model counting may not even care about such relatively tiny distinctions; it may suffice to provide rough “ball park” estimates, as long as the method is quick and the user has some confidence in its correctness. We should point out that problems with a higher solution count are not necessarily harder to determine the model count of. In fact, counters like **ReIsat** can compute the true model count of highly under-constrained problems with many “don’t care” variables and a lot of models by exploiting big clusters in the solution space. The model counting problem is instead much harder for more intricate combinatorial problems in which the solutions are spread much more finely throughout the combinatorial space.

With an abundance of difficult to count instances, scalability requirements shifted the focus on efficiency, and several techniques for fairly quickly estimating the model count have been proposed. With such estimates, one must consider two aspects: the *quality of the estimate* and the *correctness confidence* associated with the reported estimate. For example, by simply finding one solution of a formula F with a SAT solver, one can easily proclaim with high (100%) confidence that F has at least one solution—a correct lower bound on the model count. However, if F in reality has, say, 10^{15} solutions, this high confidence estimate is of very poor quality. On the other extreme, a technique may report an estimate much closer to the true count of 10^{15} , but may be completely unable to provide any correctness confidence, making one wonder how good the reported estimate actually is. We would ideally like to have some control on both the quality of the reported estimate as well as the correctness confidence associated with it. The quality may come as an empirical support for a technique in terms of it often being fairly close to the true count, while the correctness confidence may be provided in terms of convergence to the true count in the limit or as a probabilistic (or even statistical) guarantee on the reported estimate being a correct lower or upper bound. We have already mentioned in Section 25.1 one such randomized algorithm with strong theoretical guarantees, namely, the FPRAS scheme of Karp and Luby [KL85]. We discuss in the remainder of this section some approaches that have been implemented and evaluated more extensively.

25.3.1. Estimation Without Guarantees

Using sampling for estimates: Wei and Selman [WS05] introduced a local search based method that uses Markov Chain Monte Carlo (MCMC) sampling to compute an approximation of the true model count of a given formula. Their model counter, **ApproxCount**, is able to solve several instances quite accurately, while scaling much better than exact model counters as problem size increases.

ApproxCount exploits the fact that if one can sample (near-)uniformly from the set of solutions of a formula F , then one can compute a good estimate of

the number of solutions of F .⁸ The basic idea goes back to Jerrum, Valiant, and Vazirani [JVV86]. Consider a Boolean formula F with M satisfying assignments. Assuming we could sample these satisfying assignments uniformly at random, we can estimate the fraction of all models that have a variable x set to TRUE, M^+ , by taking the ratio of the number of models in the sample that have x set to TRUE over the total sample size. This fraction will converge, with increasing sample size, to the true fraction of models with x set positively, namely, $\gamma = M^+/M$. (For now, assume that $\gamma > 0$.) It follows immediately that $M = (1/\gamma)M^+$. We will call $1/\gamma$ the “multiplier” (> 0). We have thus reduced the problem of counting the models of F to counting the models of a simpler formula, $F^+ = F|_{x=\text{TRUE}}$; the model count of F is simply $1/\gamma$ times the model count of F^+ . We can recursively repeat the process, leading to a series of multipliers, until all variables are assigned truth values or—more practically—until the residual formula becomes small enough for us to count its models with an exact model counter. For robustness, one usually sets the selected variable to the truth value that occurs more often in the sample, thereby keeping intact a majority of the solutions in the residual formula and recursively counting them. This also avoids the problem of having $\gamma = 0$ and therefore an infinite multiplier. Note that the more frequently occurring truth value gives a multiplier $1/\gamma$ of at most 2, so that the estimated count grows relatively slowly as variables are assigned values.

In **ApproxCount**, the above strategy is made practical by using an efficient solution sampling method called **SampleSat** [WES04], which is an extension of the well-known local search SAT solver **Walksat** [SKC96]. Efficiency and accuracy considerations typically suggest that we obtain 20-100 samples per variable setting and after all but 100-300 variables have been set, give the residual formula to an exact model counter like **ReIsat** or **Cachet**.

Compared to exact model counters, **ApproxCount** is extremely fast and has been shown to provide very good estimates for solution counts. Unfortunately, there are no guarantees on the uniformity of the samples from **SampleSat**. It uses Markov Chain Monte Carlo (MCMC) methods [Mad02, MRR⁺53, KGV83], which often have exponential (and thus impractical) mixing times for intricate combinatorial problems. In fact, the main drawback of Jerrum et al.’s counting strategy is that for it to work well one needs uniform (or near-uniform) sampling, which is a very hard problem in itself. Moreover, biased sampling can lead to arbitrarily bad under- or over-estimates of the true count. Although the counts obtained from **ApproxCount** can be surprisingly close to the true model counts, one also observes cases where the method significantly over-estimates or under-estimates.

Interestingly, the inherent strength of most state-of-the-art SAT solvers comes actually from the ability to quickly narrow down to a certain portion of the search space the solver is designed to handle best. Such solvers therefore sample solutions in a highly non-uniform manner, making them seemingly ill-suited for model counting, unless one forces the solver to explore the full combinatorial space. An

⁸Note that a different approach, in principle, would be to estimate the *density* of solutions in the space of all 2^n truth assignments for an n -variable formula, and extrapolate that to the number of solutions. This would require sampling all truth assignments uniformly and computing how often a sampled assignment is a solution. This is unlikely to be effective in formulas of interest, which have very sparse solution spaces, and is not what **ApproxCount** does.

intriguing question (which will be addressed in Section 25.3.2) is whether there is a way around this apparent limitation of the use of state-of-the-art SAT solvers for model counting.

Using importance sampling: Gogate and Dechter [GD07] proposed a model counting technique called **SampleMinisat**, which is based on sampling from the so-called backtrack-free search space of a Boolean formula through **SampleSearch** [GD06]. They use an importance re-sampling method at the base. Suppose we wanted to sample all solutions of F uniformly at random. We can think of these solutions sitting at the leaves of a DPLL search tree for F . Suppose this search tree has been made backtrack-free, i.e., all branches that do not lead to any solution have been pruned away. (Of course, this is likely to be impractical to achieve perfectly in practice in reasonably large and complex solution spaces without spending an exponential amount of time constructing the complete tree, but we will attempt to approximate this property.) In this backtrack-free search space, define a random process that starts at the root and at every branch chooses to follow either child with equal probability. This yields a probability distribution on all satisfying assignments of F (which are the leaves of this backtrack-free search tree), assigning a probability of 2^{-d} to a solution if there are d branching choices in the tree from the root to the corresponding leaf. In order to sample not from this particular “backtrack-free” distribution but from the *uniform* distribution over all solutions, one can use the importance sampling technique [Rub81], which works as follows. First sample k solutions from the backtrack-free distribution, then assign a new probability to each sampled solution which is proportional to the inverse of its original probability in the backtrack-free distribution (i.e., proportional to 2^d), and finally sample one solution from this new distribution. As k increases, this process, if it could be made practical, provably converges to sampling solutions uniformly at random.

SampleMinisat builds upon this idea, using DPLL-based SAT solvers to construct the backtrack-free search space, either completely or to an approximation. A simple modification of the above uniform sampling method can be used to instead estimate the *number* of solutions (i.e., the number of leaves in the backtrack-free search space) of F . The process is embedded inside a DPLL-based solver, which keeps track of which branches of the search tree have already been shown to be unsatisfiable. As more of the search tree is explored to generate samples, more branches are identified as unsatisfiable, and one gets closer to achieving the exact backtrack-free distribution. In the limit, as the number of solution samples increases to infinity, the entire search tree is explored and all unsatisfiable branches tagged, yielding the true backtrack-free search space. Thus, this process in the limit converges to purely uniform solutions and an accurate estimate of the number of solutions. Experiments with **SampleMinisat** show that it can provide very good estimates of the solution count when the formula is within the reach of DPLL-based methods. In contrast, **ApproxCount** works well when the formula is more suitable for local search techniques like **Walksat**.

Gogate and Dechter [GD07] show how this process, when using the exact backtrack-free search space (as opposed to its approximation), can also be used to provide lower bounds on the model count with probabilistic correctness guarantees following the framework of **SampleCount**, which we discuss next.

25.3.2. Lower and Upper Bounds With Guarantees

Using sampling for estimates with guarantees: Building upon `ApproxCount`, Gomes et al. [GHSS07a] showed that, somewhat surprisingly, using sampling with a modified, randomized strategy, one can get provable *lower bounds* on the total model count, with high confidence (probabilistic) correctness guarantees, *without any requirement on the quality of the sampling process*. They provide a formal analysis of the approach, implement it as the model counter `SampleCount`, and experimentally demonstrate that it provides very good lower bounds—with high confidence and within minutes—on the model counts of many problems which are completely out of reach of the best exact counting methods. The key feature of `SampleCount` is that the correctness of the bound reported by it holds even when the sampling method used is arbitrarily bad; only the quality of the bound may deteriorate (i.e., the bound may get farther away from the true count on the lower side). Thus, the strategy remains sound even when a heuristic-based practical solution sampling method is used instead of a true uniform sampler.

The idea is the following. Instead of using the solution sampler to select the variable setting and to compute a multiplier, let us use the sampler only as a heuristic to determine *in what order* to set the variables. In particular, we will use the sampler to select a variable whose positive and negative setting occurs most balanced in our set of samples (ties are broken randomly). Note that such a variable will have the highest possible multiplier (closest to 2) in the `ApproxCount` setting discussed above. Informally, setting the most balanced variable will divide the solution space most evenly (compared to setting one of the other variables). Of course, our sampler may be heavily biased and we therefore cannot really rely on the observed ratio between positive and negative settings of a variable. Interestingly, we can simply set the variable to a randomly selected truth value and use the multiplier 2. This strategy will still give—in expectation—the true model count. A simple example shows why this is so. Consider the formula F used in the discussion in Section 25.3.1 and assume x occurs most balanced in our sample. Let the model count of F^+ be $2M/3$ and of F^- be $M/3$. If we decide with probability $1/2$ to set x to `TRUE`, we obtain a total model count of $2 \times 2M/3$, i.e., too high; but, with probability $1/2$, we will set x to `FALSE`, obtaining a total count of $2 \times M/3$, i.e., too low. Together, these imply an expected (average) count of exactly M .

Technically, the *expected total model count* is correct because of the linearity of expectation. However, we also see that we may have significant *variance* between specific counts, especially since we are setting a series of variables in a row (obtaining a sequence of multipliers of 2), until we have a simplified formula that can be counted exactly. In fact, in practice, the distribution of the estimated total count (over different runs) is often heavy-tailed [KSTW06]. To mitigate the fluctuations between runs, we use our samples to select the “best” variables to set next. Clearly, a good heuristic would be to set such “balanced” variables first. We use `SampleSat` to get guidance on finding such balanced variables. The random value setting of the selected variable leads to an expected model count that is equal to the actual model count of the formula. Gomes et al. [GHSS07a] show how this property can be exploited using Markov’s inequality to obtain lower

bounds on the total model count with predefined confidence guarantees. These guarantees can be made arbitrarily strong by repeated iterations of the process.

What if all variable are found to be not so balanced? E.g., suppose the most balanced variable x is TRUE in 70% of the sampled solutions and FALSE in the remaining 30%. While one could still set x to TRUE or FALSE with probability $1/2$ each as discussed above and still maintain a correct count in expectation, Kroc et al. [KSS08] discuss how one might reduce the resulting variance by instead using a *biased coin* with probability $p = 0.7$ of Heads, setting x to TRUE with probability 0.7, and scaling up the resulting count by $1/0.7$ if x was set to TRUE and by $1/0.3$ if x was set to FALSE. If the samples are uniform, this process provably reduces the variance, compared to using an unbiased coin with $p = 0.5$.

The effectiveness of `SampleCount` is further boosted by using variable “equivalence” when no single variable appears sufficiently balanced in the sampled solutions. For instance, if variables x_1 and x_2 occur with the same polarity (either both TRUE or both FALSE) in nearly half the sampled solutions and with a different polarity in the remaining, we randomly replace x_2 with either x_1 or $\neg x_1$, and simplify. This turns out to have the same simplification effect as setting a single variable, but is more advantageous when no single variable is well balanced.

Using XOR-streamlining: `MBound` [GSS06] is a very different method for model counting, which interestingly uses any complete SAT solver “as is” in order to compute an estimate of the model count of the given formula. It follows immediately that the more efficient the SAT solver used, the more powerful this counting strategy becomes. `MBound` is inspired by work on so-called “streamlining constraints” [GS04], in which additional, non-redundant constraints are added to the original problem to increase constraint propagation and to focus the search on a small part of the subspace, (hopefully) still containing solutions. This technique was earlier shown to be successful in solving very hard combinatorial design problems, with carefully created, domain-specific streamlining constraints. In contrast, `MBound` uses a domain-independent streamlining process, where the streamlining constraints are constructed purely at random.

The central idea of the approach is to use a special type of randomly chosen constrains, namely XOR or parity constraints on the original variables of the problem. Such constraints require that an *odd* number of the involved variables be set to TRUE. (This requirement can be translated into the usual CNF form by using additional variables [Tse68], and can also be modified into requiring that an *even* number of variables be TRUE by adding the constant 1 to the set of involved variables.)

`MBound` works in a very simple fashion: repeatedly add a number s of purely random XOR constraints to the formula as additional CNF clauses, feed the resulting streamlined formula to a state-of-the-art complete SAT solver without any modification, and record whether or not the streamlined formula is still satisfiable. At a very high level, each random XOR constraint will cut the solution space of satisfying assignments approximately in half. As a result, intuitively speaking, if after the addition of s XOR constraints the formula is still satisfiable, the original formula must have at least of the order of 2^s models. More rigorously, it can be shown that if we perform t experiments of adding s random XOR constraints and our formula remains satisfiable in each case, then with probability at least

$1 - 2^{-\alpha t}$, our original formula will have at least $2^{s-\alpha}$ satisfying assignments for any $\alpha > 0$, thereby obtaining a lower bound on the model count with a probabilistic correctness guarantee. The confidence expression $1 - 2^{-\alpha t}$ says that by repeatedly doing more experiments (by increasing t) or by weakening the claimed bound of $2^{s-\alpha}$ (by increasing α), one can arbitrarily boost the confidence in the lower bound count reported by this method.

The method generalizes to the case where some streamlined formulas are found to be satisfiable and some are not. Similar results can also be derived for obtaining an upper bound on the model count, although the variance-based analysis in this case requires that the added XOR constraints involve as many as $n/2$ variables on average, which often decreases the efficiency of SAT solvers on the streamlined formula.⁹ The efficiency of the method can be further boosted by employing a *hybrid* strategy: instead of feeding the streamlined formula to a SAT solver, feed it to an exact model counter. If the exact model counter finds M solutions to the streamlined formula, the estimate of the total model count of the original formula is taken to be $2^{s-\alpha} \times M$, again with a formal correctness probability attached to statistics over this estimate over several iterations; the minimum, maximum, and average values over several iterations result in different associated correctness confidence.

A surprising feature of this approach is that it does not depend at all on how the solutions are distributed throughout the search space. It relies on the very special properties of random parity constraints, which in effect provide a good hash function, randomly dividing the solutions into two near-equal sets. Such constraints were earlier used by Valiant and Vazirani [VV86] in a randomized reduction from SAT to the related problem UniqueSAT (a “promise problem” where the input is guaranteed to have either 0 or 1 solution, never 2 or more), showing that UniqueSAT is essentially as hard as SAT. Stockmeyer [Sto85] had also used similar ideas under a theoretical setting. The key to converting this approach into a state-of-the-art model counter was the relatively recent observation that very short XOR constraints—the only ones that are practically feasible with modern constraint solvers—can provide a fairly accurate estimate of the model count on a variety of domains of interest [GSS06, GHSS07b].

Exploiting Belief Propagation methods: Kroc et al. [KSS08] showed how one can use probabilistic reasoning methods for model counting. Their algorithm, called **BPCount**, builds upon the lower bounding framework of **SampleCount**. However, instead of using several solution samples for heuristic guidance—which can often be time consuming—they use a probabilistic reasoning approach called Belief Propagation or BP. A description of BP methods in any reasonable detail is beyond the scope of this chapter; we refer the reader to standard texts on this subject [e.g. Pea88] as well as to Part 2, Chapter 22 of this Handbook. In essence, BP is a general “message passing” procedure for probabilistic reasoning, and is often described in terms of a set of mutually recursive equations which are solved in an iterative manner. On SAT instances, BP works by passing likelihood

⁹It was later demonstrated empirically [GHSS07b] that for several problem domains, significantly shorter XOR constraints are effectively as good as XOR constraints of length $n/2$ (“full length”) in terms of the key property needed for the accuracy of this method: low *variance* in the solution count estimate over several runs of **MBound**.

information between variables and clauses in an iterative fashion until a fixed point is reached. From this fixed point, statistical information about the solution space of the formula can be easily computed. This statistical information is provably exact when the constraint graph underlying the formula has a tree-like structure, and is often reasonably accurate (empirically) even when the constraint graph has cycles [cf. MWJ99].

For our purposes, BP, in principle, provides *precisely* the information deduced from solution samples in **SampleCount**, namely, an estimate of the *marginal probability* of each variable being TRUE or FALSE when all solutions are sampled uniformly at random. Thus, **BPCount** works exactly like **SampleCount** and provides the same probabilistic correctness guarantees, but is often orders of magnitude faster on several problem domains because running BP on the formula is often much faster than obtaining several solution samples through **SampleSat**. A challenge in this approach is that the standard mutually recursive BP equations don't even converge to any fixed point on many practical formulas of interest. To address this issue, Kroc et al. [KSS08] employ a *message-damping* variant of BP whose convergence behavior can be controlled by a continuously varying parameter. They also use *safety checks* in order to avoid fatal mistakes when setting variables (i.e., to avoid inadvertently making the formula unsatisfiable).

We note that the model count of a formula can, in fact, also be estimated directly from just one fixed point run of the BP equations, by computing the value of so-called partition function [YFW05]. However, the count estimated this way is often highly inaccurate on structured loopy formulas. **BPCount**, on the other hand, makes a much more robust use of the information provided by BP.

Statistical upper bounds: In a different direction, Kroc et al. [KSS08] propose a second method, called **MiniCount**, for providing *upper bounds* on the model counts of formulas. This method exploits the power of modern DPLL-based SAT solvers, which are extremely good at finding single solutions to Boolean formulas through backtrack search. The problem of computing upper bounds on the model count had so far eluded solution because of the asymmetry (discussed earlier in the partial counts sub-section of Section 25.2.1) which manifests itself in at least two inter-related forms: the set of solutions of interesting n variable formulas typically forms a minuscule fraction of the full space of 2^n truth assignments, and the application of Markov's inequality as in the lower bound analysis of **SampleCount** does not yield interesting upper bounds. As noted earlier, this asymmetry also makes upper bounds provided by partial runs of exact model counters often not very useful. To address this issue, Kroc et al. [KSS08] develop a *statistical framework* which lets one compute upper bounds under certain statistical assumptions, which are independently validated.

Specifically, they describe how the SAT solver **MiniSat** [ES05], with two minor modifications—randomizing whether the chosen variable is set to TRUE or to FALSE, and disabling restarts—can be used to estimate the total number of solutions. The number d of branching decisions (not counting unit propagations and failed branches) made by **MiniSat** before reaching a solution, is the main quantity of interest: when the choice between setting a variable to TRUE or to

FALSE is randomized,¹⁰ the number d is provably not any lower, in expectation, than \log_2 of the true model count. This provides a strategy for obtaining upper bounds on the model count, only if one could efficiently estimate the expected value, $\mathbb{E}[d]$, of the number of such branching decisions. A natural way to estimate $\mathbb{E}[d]$ is to perform multiple runs of the randomized solver, and compute the average of d over these runs. However, if the formula has many “easy” solutions (found with a low value of d) and many “hard” solutions (requiring large d), the limited number of runs one can perform in a reasonable amount of time may be insufficient to hit many of the “hard” solutions, yielding too low of an estimate for $\mathbb{E}[d]$ and thus an incorrect upper bound on the model count.

Interestingly, they show that for many families of formulas, d has a distribution that is very close to the normal distribution; in other words, the estimate 2^d of the upper bound on the number of solutions is log-normally distributed. Now, under the assumption that d is indeed normally distributed, estimating $\mathbb{E}[d]$ for an upper bound on the model count becomes easier: when sampling various values of d through multiple runs of the solver, one need not necessarily encounter both low and high values of d in order to correctly estimate $\mathbb{E}[d]$. Instead, even with only below-average samples of d , the assumption of normality lets one rely on standard statistical tests and conservative computations to obtain a statistical upper bound on $\mathbb{E}[d]$ within any specified confidence interval. We refer the reader to the full paper for details. Experimentally, `MiniCount` is shown to provide good upper bounds on the solution counts of formulas from many domains, often within seconds and fairly close to the true counts (if known) or separately computed lower bounds.

25.4. Conclusion

With SAT solving establishing its mark as one of the most successful automated reasoning technologies, the model counting problem for SAT has begun to see a surge of activity in the last few years. One thing that has been clear from the outset is that model counting is a much harder problem. Nonetheless, thanks to its broader scope and applicability than SAT solving, it has led to a range of new ideas and approaches—from DPLL-based methods to local search sampling estimates to knowledge compilation to novel randomized streamlining methods. Practitioners working on model counting have discovered that many interesting techniques that were too costly for SAT solving are not only cost effective for model counting but crucial for scaling practical model counters to reasonably large instances. The variety of tools for model counting is already rich, and growing. While we have made significant progress, at least two key challenges remain open: how do we push the limits of scalability of model counters even further, and how do we extend techniques to do model counting for *weighted* satisfiability problems?¹¹ While exact model counters are often able to also solve

¹⁰`MiniSat` by default always sets variables to FALSE.

¹¹In weighted model counting, each variable x has a weight $p(x) \in [0, 1]$ when set to TRUE and a weight $1 - p(x)$ when set to FALSE. The weight of a truth assignment is the product of the weights of its literals. The weighted model count of a formula is the sum of the weights of its satisfying assignments.

the weighted version of the problem at no extra cost, much work needs to be done to adapt the more scalable approximate methods to handle weighted model counting.

References

- [Ang80] D. Angluin. On counting problems and the polynomial-time hierarchy. *Theoretical Computer Science*, 12:161–173, 1980.
- [BDK07] D. D. Bailey, V. Dalmau, and P. G. Kolaitis. Phase transitions of PP-complete satisfiability problems. *Discrete Applied Mathematics*, 155(12):1627–1639, 2007.
- [BDP03] F. Bacchus, S. Dalmau, and T. Pitassi. Algorithms and complexity results for #SAT and Bayesian inference. In *Proceedings of FOCS-03: 44th Annual Symposium on Foundations of Computer Science*, pages 340–351, Cambridge, MA, October 2003.
- [BIPS03] P. Beame, R. Impagliazzo, T. Pitassi, and N. Segerlind. Memoization and DPLL: Formula caching proof systems. In *Proceedings 18th Annual IEEE Conference on Computational Complexity*, pages 225–236, Aarhus, Denmark, July 2003.
- [BL99] E. Birnbaum and E. L. Lozinskii. The good old Davis-Putnam procedure helps counting models. *Journal of Artificial Intelligence Research*, 10:457–477, 1999.
- [BP00] R. J. Bayardo Jr. and J. D. Pehoushek. Counting models using connected components. In *Proceedings of AAAI-00: 17th National Conference on Artificial Intelligence*, pages 157–162, Austin, TX, 2000.
- [Bry86] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
- [BS97] R. J. Bayardo Jr. and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of AAAI-97: 14th National Conference on Artificial Intelligence*, pages 203–208, Providence, RI, July 1997.
- [Dar01] A. Darwiche. Decomposable negation normal form. *Journal of the ACM*, 48(4):608–647, 2001.
- [Dar04] A. Darwiche. New advances in compiling CNF into decomposable negation normal form. In *Proceedings of ECAI-04: 16th European Conference on Artificial Intelligence*, pages 328–332, Valencia, Spain, August 2004.
- [Dar05] A. Darwiche. The quest for efficient probabilistic inference, July 2005. Invited Talk, IJCAI-05.
- [DB07] J. Davies and F. Bacchus. Using more reasoning to improve #SAT solving. In *Proceedings of AAAI-07: 22nd National Conference on Artificial Intelligence*, pages 185–190, Vancouver, BC, July 2007.
- [DM02] A. Darwiche and P. Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.
- [ES05] N. Eén and N. Sörensson. MiniSat: A SAT solver with conflict-clause minimization. In *Proceedings of SAT-05: 8th International Conference on Theory and Applications of Satisfiability Testing*, St. Andrews, U.K., June 2005.

- [FMR08] E. Fischer, J. A. Makowsky, and E. V. Ravve. Counting truth assignments of formulas of bounded tree-width or clique-width. *Discrete Applied Mathematics*, 156(4):511–529, 2008.
- [GD06] V. Gogate and R. Dechter. A new algorithm for sampling CSP solutions uniformly at random. In *CP-06: 12th International Conference on Principles and Practice of Constraint Programming*, volume 4204 of *Lecture Notes in Computer Science*, pages 711–715, Nantes, France, September 2006.
- [GD07] V. Gogate and R. Dechter. Approximate counting by sampling the backtrack-free search space. In *Proceedings of AAAI-07: 22nd National Conference on Artificial Intelligence*, pages 198–203, Vancouver, BC, July 2007.
- [GHSS07a] C. P. Gomes, J. Hoffmann, A. Sabharwal, and B. Selman. From sampling to model counting. In *Proceedings of IJCAI-07: 20th International Joint Conference on Artificial Intelligence*, pages 2293–2299, Hyderabad, India, January 2007.
- [GHSS07b] C. P. Gomes, J. Hoffmann, A. Sabharwal, and B. Selman. Short XORs for model counting; from theory to practice. In *Proceedings of SAT-07: 10th International Conference on Theory and Applications of Satisfiability Testing*, volume 4501 of *Lecture Notes in Computer Science*, pages 100–106, Lisbon, Portugal, May 2007.
- [GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.
- [GS04] C. P. Gomes and M. Sellmann. Streamlined constraint reasoning. In *CP-04: 10th International Conference on Principles and Practice of Constraint Programming*, volume 3258 of *Lecture Notes in Computer Science*, pages 274–289, Toronto, Canada, October 2004.
- [GSS02] G. Gottlob, F. Scarcello, and M. Sideri. Fixed-parameter complexity in AI and nonmonotonic reasoning. *Artificial Intelligence*, 138(1-2):55–86, 2002.
- [GSS06] C. P. Gomes, A. Sabharwal, and B. Selman. Model counting: A new strategy for obtaining good bounds. In *Proceedings of AAAI-06: 21st National Conference on Artificial Intelligence*, pages 54–61, Boston, MA, July 2006.
- [JVV86] M. R. Jerrum, L. G. Valiant, and V. V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theoretical Computer Science*, 43:169–188, 1986.
- [KGV83] S. Kirkpatrick, D. Gelatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.
- [KL85] R. M. Karp and M. Luby. Monte-Carlo algorithms for the planar multiterminal network reliability problem. *Journal of Complexity*, 1(1):45–64, 1985.
- [KLM89] R. M. Karp, M. Luby, and N. Madras. Monte-Carlo approximation algorithms for enumeration problems. *Journal of Algorithms*, 10(3):429–448, 1989.
- [KSS08] L. Kroc, A. Sabharwal, and B. Selman. Leveraging belief propagation,

- backtrack search, and statistics for model counting. In *CPAIOR-08: 5th International Conference on Integration of AI and OR Techniques in Constraint Programming*, volume 5015 of *Lecture Notes in Computer Science*, pages 127–141, Paris, France, May 2008.
- [KSTW06] P. Kilby, J. Slaney, S. Thiébaux, and T. Walsh. Estimating search tree size. In *Proceedings of AAAI-06: 21st National Conference on Artificial Intelligence*, pages 1014–1019, Boston, MA, July 2006.
- [LMP01] M. L. Littman, S. M. Majercik, and T. Pitassi. Stochastic Boolean satisfiability. *Journal of Automated Reasoning*, 27(3):251–296, 2001.
- [Mad02] N. Madras. Lectures on Monte Carlo methods. In *Field Institute Monographs*, volume 16. American Mathematical Society, 2002.
- [ML98] S. M. Majercik and M. L. Littman. Using caching to solve larger probabilistic planning problems. In *Proceedings of AAAI-98: 15th National Conference on Artificial Intelligence*, pages 954–959, Madison, WI, July 1998.
- [MMZ⁺01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of DAC-01: 38th Design Automation Conference*, pages 530–535, Las Vegas, NV, June 2001.
- [MRR⁺53] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equations of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.
- [MWJ99] K. Murphy, Y. Weiss, and M. Jordan. Loopy belief propagation for approximate inference: An empirical study. In *Proceedings of UAI-99: 15th Conference on Uncertainty in Artificial Intelligence*, pages 467–475, Sweden, July 1999.
- [NRS06] N. Nishimura, P. Ragde, and S. Szeider. Solving #SAT using vertex covers. In *Proceedings of SAT-06: 9th International Conference on Theory and Applications of Satisfiability Testing*, volume 4121 of *Lecture Notes in Computer Science*, pages 396–409, Seattle, WA, 2006.
- [Pap94] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
- [Par02] J. D. Park. MAP complexity results and approximation methods. In *Proceedings of UAI-02: 18th Conference on Uncertainty in Artificial Intelligence*, pages 388–396, Edmonton, Canada, August 2002.
- [PD07] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Proceedings of SAT-07: 10th International Conference on Theory and Applications of Satisfiability Testing*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299, Lisbon, Portugal, May 2007.
- [Pea88] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.
- [Rot96] D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1-2):273–302, 1996.
- [Rub81] R. Y. Rubinstein. *Simulation and the Monte Carlo Method*. John Wiley & Sons, 1981.
- [SBB⁺04] T. Sang, F. Bacchus, P. Beame, H. A. Kautz, and T. Pitassi. Combin-

- ing component caching and clause learning for effective model counting. In *Proceedings of SAT-04: 7th International Conference on Theory and Applications of Satisfiability Testing*, Vancouver, BC, 2004.
- [SBK05a] T. Sang, P. Beame, and H. A. Kautz. Heuristics for fast exact model counting. In *Proceedings of SAT-05: 8th International Conference on Theory and Applications of Satisfiability Testing*, volume 3569 of *Lecture Notes in Computer Science*, pages 226–240, St. Andrews, U.K., June 2005.
- [SBK05b] T. Sang, P. Beame, and H. A. Kautz. Performing Bayesian inference by weighted model counting. In *Proceedings of AAAI-05: 20th National Conference on Artificial Intelligence*, pages 475–482, Pittsburgh, PA, July 2005.
- [SKC96] B. Selman, H. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In D. S. Johnson and M. A. Trick, editors, *Cliques, Coloring and Satisfiability: the Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 521–532. American Mathematical Society, 1996.
- [Sto85] L. J. Stockmeyer. On approximation algorithms for $\#P$. *SIAM Journal on Computing*, 14(4):849–861, 1985.
- [Thu06] M. Thurley. sharpSAT - counting models with advanced component caching and implicit BCP. In *Proceedings of SAT-06: 9th International Conference on Theory and Applications of Satisfiability Testing*, volume 4121 of *Lecture Notes in Computer Science*, pages 424–429, Seattle, WA, August 2006.
- [Tod89] S. Toda. On the computational power of PP and $\oplus P$. In *FOCS-89: 30th Annual Symposium on Foundations of Computer Science*, pages 514–519, 1989.
- [Tse68] G. S. Tseitin. On the complexity of derivation in the propositional calculus. In A. O. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logic, Part II*. 1968.
- [Val79] L. G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201, 1979.
- [VV86] L. G. Valiant and V. V. Vazirani. NP is as easy as detecting unique solutions. *Theoretical Computer Science*, 47(3):85–93, 1986.
- [WES04] W. Wei, J. Erenrich, and B. Selman. Towards efficient sampling: Exploiting random walk strategies. In *Proceedings of AAAI-04: 19th National Conference on Artificial Intelligence*, pages 670–676, San Jose, CA, July 2004.
- [WS05] W. Wei and B. Selman. A new approach to model counting. In *Proceedings of SAT-05: 8th International Conference on Theory and Applications of Satisfiability Testing*, volume 3569 of *Lecture Notes in Computer Science*, pages 324–339, St. Andrews, U.K., June 2005.
- [YFW05] J. S. Yedidia, W. T. Freeman, and Y. Weiss. Constructing free-energy approximations and generalized belief propagation algorithms. *IEEE Transactions on Information Theory*, 51(7):2282–2312, 2005.

Chapter 26

Approximate Model Counting

Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi

26.1. Introduction

Model counting, or counting the solutions to a set of constraints, is a fundamental computational problem with applications in diverse areas spanning probabilistic inference, network reliability estimation, statistical physics, explainable AI, security vulnerability analysis, and the like [Rot96, DH07, XCD12, ABB15, DOMPV17, LSS17, BEH⁺18, ZQRZ18, NSM⁺19, BSS⁺19]. While exact counting is known to be computationally hard (see Chapter 25 for more details on exact counting), a large class of applications can benefit even if approximate model counts are available. Approximate counting algorithms have therefore been studied in great depth by both theoreticians and practitioners. In this chapter, we discuss some of the key ideas behind modern approximate model counters for propositional constraints.

We begin with some notation and terminology. Let φ be a propositional formula. We use $|\varphi|$ to denote the size of φ presented as a string to a Turing machine, and call the set of all variables in φ the *support* of φ , or $\text{Sup}(\varphi)$. Following standard terminology, a variable or its negation is called a *literal*, a *clause* is a disjunction of literals, and a *cube* is a conjunction of literals. We say φ is in *conjunctive normal form* (CNF) if it is a conjunction of clauses. Similarly, we say φ is in *disjunctive normal form* (DNF) if it is a disjunction of cubes. A satisfying assignment, solution, or model of φ is an assignment of truth values to all variables in $\text{Sup}(\varphi)$ such that φ evaluates to True. We use $\text{Sol}(\varphi)$ to denote the set of all models of φ . The *model counting* problem asks: *Given φ , find $|\text{Sol}(\varphi)|$.*

26.1.1. A historical perspective

Approximate model counting techniques, at least for CNF formulas, have mostly evolved hand-in-hand with techniques for exact model counting, except in the last decade or so, when a few innovations triggered a large body of work in the broad area of approximate counting and sampling. The real value-addition of approximate counters has always been in cases that lie beyond the reach of exact counters, given computational resource constraints. Scalability has therefore been

a key consideration throughout the history of the development of approximate model counters. It was widely believed until recently (circa 2006) that this scalability can be achieved only by weakening or sacrificing approximation guarantees. As a consequence, most applications used approximate counting techniques with weak or no guarantees at all. An example of such usage is that of Monte Carlo estimators for model counting where the number of Monte Carlo steps is truncated heuristically before the theory-mandated mixing of states happen [WS05]. Fortunately, work done over the last decade has convincingly demonstrated that by striking a fine balance between theoretical and practical considerations, it is indeed possible to design approximate model counters that provide strong guarantees while also scaling to problem instances that are orders of magnitude larger than what was possible earlier. To get a sense of where we stand in 2019, state-of-the-art approximate model counters can solve problem instances with half a million propositional variables in less than 2 hours on a single high-end computing node, yielding estimates that are provably within a factor of 1.8 of the true count with at least 99% confidence [SM19].

Some of the key theoretical results pertaining to approximate counting were already known as early as the mid 1980s [Sto83, JVV86]. In a seminal paper [Sto83], Stockmeyer used universal hash functions [CW77] to solve the approximate counting problem using polynomially many NP oracle calls. While theoretically elegant, the proposed technique suffered from the requirement of a prohibitively large number of invocations of NP oracle to perform symbolic reasoning on formulas with very large numbers of variables. Jerrum, Valiant, and Vazirani demonstrated inter-reducibility of approximate counting and almost-uniform sampling in [JVV86]. Almost a decade later, Bellare, Goldreich, and Petrank reported an almost uniform generator of models using hash-functions of high degree of universality [BGP00]. Technically, this could be used to approximately count models using Jerrum et al.'s result [JVV86]. Unfortunately, our attempt in this direction [CMV13a] showed that this approach did not scale beyond formulas with more than 20 propositional variables. The major stumbling block was the requirement of inverting a hash function with a high degree of universality.

After a long period of relatively incremental improvements in approximate counting techniques, in 2006, Gomes, Sabharwal and Selman [GSS06] reported a parameterized approximate counting algorithm that used random parity constraints as universal hash functions and harnessed the power of a state-of-the-art CNF satisfiability solver in the backend. They showed that if the right combination of parameters was chosen by the user (this is not always easy), this approach could solve problems much larger than what was possible earlier, and yet provide guarantees on the quality of the estimate. Subsequently, this work was developed further in [GHSS07b, GHSS07a]. In 2013, two independent papers [CMV13b, EGSS13b] reported highly scalable approximate counting algorithms in two different settings, guaranteeing $(1 + \varepsilon)$ -factor approximation with confidence at least $1 - \delta$ for any user-provided values of ε and δ . The dependence on the user to provide the right combination of parameters was removed in these algorithms and their implementations benefitted from the then newly-proposed SAT solver *CryptoMiniSat* that could handle parity constraints natively [SNC09].

Since then, there has been a large body of work in this area, as detailed later in the chapter.

Interestingly, the history of development of approximate model counters for DNF formulas followed a different trajectory. As early as 1983, Karp and Luby showed that it is possible to approximate the model count of DNF formulas efficiently and with strong guarantees using Monte Carlo sampling [KL83]. While there have been several follow-up works that built and improved on this algorithm, the success of these techniques could not be lifted to the case of CNF formulas due to key steps that exploit the fact that the input formula is in DNF. Monte Carlo techniques have dominated the landscape of approximate counters for DNF formulas since Karp and Luby’s early work. It is only recently that an alternative approach using universal hash functions has been found to achieve the same guarantees and the same asymptotic time complexity [CMV16, MSV17, MSV19] as the best-known Monte Carlo algorithm. As observed, however, in [MSV19], the empirically observed runtime of approximate DNF counting algorithms is much more nuanced and hard to predict.

26.1.2. An overview of the complexity landscape

Exact model counting, and even some variants of approximate counting are computationally hard in a formal sense. To appreciate this better, we review a few concepts from complexity theory. The reader is assumed to be familiar with the notions of Turing machines, as well as the complexity classes P , NP , $coNP$ and their generalization to the polynomial hierarchy (PH). A detailed exposition on these can be found in [AB09]. Given an instance of a problem in NP , the corresponding counting problem asks how many accepting paths exist in the non-deterministic Turing machine that solves the problem. The complexity class $\#P$ is defined to be the set of counting problems associated with all decision problems in NP . Toda showed that a single invocation of a $\#P$ -oracle suffices to solve every problem in the polynomial hierarchy efficiently [Tod89]. Formally, $PH \subseteq P^{\#P}$, where $\mathcal{C}_1^{\mathcal{C}_2}$ denotes the class of problems that can be solved by a Turing machine for a problem in \mathcal{C}_1 when equipped with an oracle for a complete problem in \mathcal{C}_2 . A consequence of Toda’s theorem is that problems in $\#P$ (viz. exact model counting) are likely much harder than those in PH.

A *probabilistic Turing machine* is a non-deterministic Turing machine with specially designated “coin-tossing” states. When the machine reaches one of these states, it can choose between alternative transitions based on a probability distribution of coin tosses. For our purposes, the probability distribution is assumed to be uniform. Computing with probabilistic Turing machines gives rise to probabilistic complexity classes like RP , BPP etc. The reader is referred to [AB09] for more details on these classes. In a functional problem like model counting, for every input x , we seek an answer $f(x) \in \mathbb{N}$ of size polynomial in $|x|$. A *fully polynomial randomized approximation scheme* (FPRAS) for such a problem is a probabilistic Turing machine that takes as input x and a parameter $\varepsilon (> 0)$. It then generates an estimate c that lies within $f(x) \cdot (1 + \varepsilon)^{-1}$ and $f(x) \cdot (1 + \varepsilon)$ with probability strictly greater than $\frac{1}{2}$, while taking time polynomial in $|x|$ and $\frac{1}{\varepsilon}$. Given an FPRAS for a functional problem, the probability of generating an estimate outside the tolerance window can be reduced below any desired confidence

threshold δ ($0 < \delta \leq 1$) by running the FPRAS $\mathcal{O}(\log \frac{1}{\delta})$ times independently and by choosing the median estimate.

Valiant showed that exact model counting is $\#\text{P}$ -complete for both CNF and DNF formulas [Val79]. Therefore, it is unlikely that any efficient algorithm exists for exact model counting. This has motivated significant research on different variants of approximate model counting over the past four decades. To have a unified view of these variants, we introduce some notation. Let Z be a random variable with a specified probability distribution. Let $\bowtie \in \{=, \leq, <, >, \geq\}$ be a relational operator and let a be a value in the range of Z . We use $\Pr[Z \bowtie a]$ to denote the probability that $Z \bowtie a$ holds. We also use $\mathbb{E}[Z]$ and $\text{Var}[Z]$ to denote the expectation and variance, respectively, of Z . An approximate model counter takes as input a formula φ (along with possibly tolerance and confidence parameters) and returns an estimate c that approximates $|\text{Sol}(\varphi)|$. Depending on whether the algorithm is deterministic or randomized, and depending on the type of approximation used, we can identify several variants of approximate model counters. In the following discussion, unless specified otherwise, ε (> 0) represents a (additive or multiplicative) tolerance used in approximating $|\text{Sol}(\varphi)|$, and δ ($0 < \delta \leq 1$) represents a confidence bound for a randomized algorithm.

A *deterministic additive approximate model counter* takes a formula φ and an additive tolerance ε (> 0) as inputs, and returns an estimate c (≥ 0) such that $(|\text{Sol}(\varphi)| - \varepsilon) \leq c \leq (|\text{Sol}(\varphi)| + \varepsilon)$. A *probabilistic additive approximate model counter* takes an additional confidence parameter δ ($0 < \delta \leq 1$) as input, and finds a random estimate c (≥ 0) such that $\Pr[(|\text{Sol}(\varphi)| - \varepsilon) \leq c \leq (|\text{Sol}(\varphi)| + \varepsilon)] \geq 1 - \delta$. Unfortunately, if $\varepsilon \leq 2^{\frac{|\text{Sup}(\varphi)|}{2} - 2}$, both versions of additive approximate model counting are computationally hard. Specifically, the deterministic version continues to be $\#\text{P}$ -complete, while the probabilistic version lies beyond PH, unless the entire PH collapses [CMV19]. Thus, additive approximations do not simplify the problem of counting from a worst-case complexity perspective, and we choose not to elaborate further on this class of counters. It is worth mentioning, however, that approximate counters with additive tolerance have been used in some applications, viz. [SVP⁺16] and [FHO13].

Counters with multiplicative approximations have been much more extensively studied. A *deterministic multiplicative approximate model counter* takes as inputs a formula φ and a multiplicative tolerance ε (> 0) and returns an estimate c (≥ 0) such that $|\text{Sol}(\varphi)| \cdot (1 + \varepsilon)^{-1} \leq c \leq |\text{Sol}(\varphi)| \cdot (1 + \varepsilon)$. A *probabilistic multiplicative approximate model counter*, also called a *probably approximately correct (PAC)* model counter, takes an additional confidence parameter δ ($0 < \delta \leq 1$) as input and returns a random estimate c (≥ 0) such that $\Pr[|\text{Sol}(\varphi)| \cdot (1 + \varepsilon)^{-1} \leq c \leq |\text{Sol}(\varphi)| \cdot (1 + \varepsilon)] \geq 1 - \delta$. In a seminal paper, Stockmeyer showed that approximate counting with multiplicative tolerance can be solved by a deterministic algorithm that makes polynomially many calls to a Σ_2^P oracle [Sto83]. Therefore, practically efficient deterministic algorithms for approximate counting can be designed if we have access to Σ_2^P solvers (also called *2QBF* solvers) that can solve large problem instances efficiently in practice. Unfortunately, despite significant advances in 2QBF solving, this has not yet yielded practically efficient deterministic approximate counters yet. Building on Stockmeyer's work, Jerrum et al. [JVV86] showed that approximate counting

with PAC guarantees can be achieved by a probabilistic Turing machine in polynomial time with access to an NP oracle. Given the spectacular improvement in performance of propositional SAT solvers over the past few decades, research in practically useful approximate model counting has largely focused on variants of approximate counters with PAC guarantees.

The variants of approximate counting discussed above require the estimate to lie within a specified tolerance window on *either* side of the exact count. The required approximation guarantees are therefore two-sided. In some applications this may not be needed, and one-sided approximation guarantees (viz. $c \geq |\text{Sol}(\varphi)| \cdot (1 + \varepsilon)^{-1}$) may be sufficient. Therefore, approximate counting with one-sided guarantees has also been studied in the literature. In this chapter, however, we focus on approximate counters with two-sided guarantees, since they provide the strongest approximation guarantees while still scaling to large problem instances in practice. The interested reader is referred to Chapter 25 for more details on counters with one-sided approximation guarantees and guarantee-less approximate counters.

A brief examination of the differences between approximate model counting for CNF and DNF formulas is in order here. It was Karp and Luby [KL83] who first showed that DNF model counting admits an FPRAS. Their algorithm and analysis were subsequently improved in [KLM89]. Recently, it was re-proved in [CMV16, MSV17] that DNF counting admits an FPRAS using techniques different from those used in [KL83, KLM89]. Unfortunately, as observed in [KL83, KL85], CNF model counting cannot admit an FPRAS unless $\text{NP} = \text{RP}$ – a longstanding unresolved question in complexity theory. The best we know at present is that approximate counting for CNF formulas with PAC guarantees can be solved by a probabilistic Turing machine *with access to an NP oracle* in time polynomial in $|\varphi|$, $\frac{1}{\varepsilon}$ and $\log \frac{1}{\delta}$. Therefore, approximate model counting for CNF and DNF formulas present different sets of challenges and we discuss them separately.

The problems of approximate counting with PAC guarantees and *almost uniform generation* of models are intimately related to each other. Given φ and a tolerance ε (≥ 0), let $\alpha(\varphi)$ be a real number in $(0, 1]$ that depends only on φ . An almost uniform generator is a probabilistic Turing machine that takes as inputs a formula φ and a tolerance ε (> 0), and outputs a random assignment π to variables in $\text{Sup}(\varphi)$ such that $\alpha(\varphi) \cdot (1 + \varepsilon)^{-1} \leq \Pr[\pi \text{ is output}] \leq \alpha(\varphi) \cdot (1 + \varepsilon)$ if π is a solution of φ , and $\Pr[\pi \text{ is output}] = 0$ otherwise. Jerrum et al. [JVV86] showed that approximate counting with PAC guarantees and almost uniform generation are polynomially inter-reducible. Thus, an almost uniform generator can be used as a sub-routine to obtain a approximate counter for φ and vice versa. Furthermore, the time-complexity of each problem is within a polynomial factor of that of the other. It turns out, however, that all practically efficient almost uniform generators developed so far already use an approximate counter (or some variant of it) internally in an intermediate step. Hence, using such a sampler to obtain an approximate counter again is not very meaningful.

26.2. Approximate Model Counting for CNF

In this section, we discuss algorithms for computing a PAC estimate of $|\text{Sol}(\varphi)|$, where φ is a CNF formula. Significantly, all state-of-the-art algorithms for this problem make use of probabilistic hash functions. Therefore, we first present a generic framework for approximate model counting using probabilistic hash functions. We then show how different algorithms developed over the years [Sto83, GSS06, CMV13b, IMMV15, AD16, CMV16, ZCSE16, AT17, AHT18, SM19] can be viewed as instantiations and optimizations of this framework.

26.2.1. Probabilistic hash-based counting

A probabilistic hash function maps elements of a given domain, viz. $\{0, 1\}^n$, to random buckets or “cells” from a finite range, viz. $\{0, 1\}^i$ where $0 \leq i \leq n$. This is achieved by choosing a hash function uniformly at random from a family of functions, each of which maps $\{0, 1\}^n$ to $\{0, 1\}^i$. In probabilistic hash-based counting, the solution space of φ is randomly partitioned into approximately equal-sized “small” cells using probabilistic hash functions. Suppose each hash function in a family has a range of 2^i cells, and we choose one of these at random. By carefully designing the hash family, we can ensure that the count of solutions of φ in an arbitrary cell of the induced partitioning of $\text{Sol}(\varphi)$ is an unbiased estimator of $\frac{|\text{Sol}(\varphi)|}{2^i}$ with small variance. Therefore, if we count the solutions of φ in a single cell and scale it by 2^i , we obtain an estimate of $|\text{Sol}(\varphi)|$.

In order to formalize the above intuition, we need some additional notation. Let $\{x_1, \dots, x_n\}$ be the support of φ . Let h_i denote a hash function that maps elements of $\{0, 1\}^n$ to $\{0, 1\}^i$, where $0 \leq i \leq n$. Let \mathcal{H}_i be a (sub-)family of such hash functions mapping $\{0, 1\}^n$ to $\{0, 1\}^i$. We use \mathcal{H} to denote the family of hash functions obtained by taking the union of all \mathcal{H}_i for $i \in \{0, \dots, n\}$.

The general structure of a probabilistic hash-based approximate counter is shown in Algorithm 1. The algorithm takes as inputs a formula φ , a tolerance bound ε (> 0) and a confidence bound δ ($0 < \delta \leq 1$). It returns an estimate c such that $\Pr [|\text{Sol}(\varphi)| \cdot (1 + \varepsilon)^{-1} \leq c \leq |\text{Sol}(\varphi)| \cdot (1 + \varepsilon)] \geq 1 - \delta$. Let us ignore lines 1 through 4 of Algorithm 1 for now. In line 5, we commit to using a family of hash functions, say \mathcal{H} . In general, \mathcal{H} has sub-families \mathcal{H}_i as discussed above. In line 8, we choose $n + 1$ hash functions, one from each sub-family \mathcal{H}_i in \mathcal{H} . Let these hash functions be $h_i : \{0, 1\}^n \rightarrow \{0, 1\}^i$ for $i \in \{0, \dots, n\}$. Depending on the counter, each h_i may be chosen independently of h_j (for $j \neq i$), or the choice of h_i may be correlated with the choice of h_j . Note that each h_i partitions $\text{Sol}(\varphi)$ into 2^i cells. Furthermore, for every $\mathbf{a} \in \{0, 1\}^i$, the set $\text{Sol}(\varphi \wedge h_i^{-1}(\mathbf{a}))$ consists of exactly those solutions of φ that are mapped to the cell \mathbf{a} by h_i .

The core of Algorithm 1 consists of the loop in lines 10–15. This loop searches for the smallest index i in $\{1, \dots, n\}$ such that the partitioning induced by h_i yields “small” cells with only a few solutions of φ per cell, with high probability. Since the choice of a cell to check for the count of solutions is arbitrary, we focus on the cell labeled all 0s, i.e. $\mathbf{0}^i$. The maximum number of solutions in a cell for it to be considered “small” is defined by a parameter `thresh` that is determined

¹We assume h_0 maps elements of $\{0, 1\}^n$ to a singleton set, chosen to be $\{0\}$.

Algorithm 1 HASHCOUNTER($\varphi, \varepsilon, \delta$)

```

1: thresh  $\leftarrow$  FINDSMALLCELLSIZE( $\varepsilon$ );            $\triangleright$  Size threshold for a cell to be considered "small"
2: smc  $\leftarrow$  SATURATINGCOUNTER( $\varphi$ , thresh);      $\triangleright$  Saturated model count of  $\varphi$ 
3: if (smc  $\neq$   $\top$ ) then return smc;                $\triangleright$  Easy case: Solution space of  $\varphi$  itself is "small"
4: itercount  $\leftarrow$  FINDREPETITIONCOUNT( $\delta$ );     $\triangleright$  # repetitions to amplify confidence
5:  $\mathcal{H} \leftarrow$  CHOOSEHASHFAMILY();
6: CountEst  $\leftarrow$  EmptyList;                      $\triangleright$  List of model count estimates from different iterations
7: repeat itercount times
8:   ( $h_0, \dots, h_n$ )  $\leftarrow$  CHOOSEHASHFUNCTIONS( $\mathcal{H}, n$ );            $\triangleright$   $n$  is # variables in  $\varphi$ 
9:   INITIALIZEARRAY(CellSize,  $n$ );            $\triangleright$  CellSize is an array of size  $n + 1$ 
    $\triangleright$  CellSize[ $i$ ] records (saturated) model count in cell  $\mathbf{0}^i$  induced by  $h_i$ ; CellSize[0] =  $\top$ 
10:  repeat until ALLINDICESASSIGNED(CellSize)
11:    $i \leftarrow$  CHOOSEUNASSIGNEDINDEX(CellSize);            $\triangleright$  Choose  $i$  s.t. CellSize[ $i$ ] =  $\perp$ 
12:   CellSize[ $i$ ]  $\leftarrow$  SATURATINGCOUNTER( $\varphi \wedge h_i^{-1}(\mathbf{0}^i)$ , thresh);
13:   if (CellSize[ $i$ ] < thresh  $\wedge$  CellSize[ $i - 1$ ] =  $\top$ ) then            $\triangleright$  Found right "small" cell
14:     Append CellSize[ $i$ ]  $\times 2^i$  to CountEst;            $\triangleright$  Current model count estimate
15:     go to line 7;            $\triangleright$  Repeat to amplify confidence
16: return COMPUTEMEDIAN(CountEst);

```

in line 1 of Algorithm 1. In general, thresh depends polynomially on $\frac{1}{\varepsilon}$, where ε is the tolerance bound of our PAC estimate. We assume access to a sub-routine `SATURATINGCOUNTER` that counts the solutions of a formula, but only upto a specified bound. Specifically, `SATURATINGCOUNTER`(ψ, c) returns a special symbol \top (denoting saturation of count) if ψ has c or more solutions. Otherwise, it returns the exact count of solutions of ψ . Thus, `SATURATINGCOUNTER`(ψ, c) always returns a value in $\{0, c - 1\} \cup \{\top\}$. We call this a *saturated count* of the solutions of ψ . Algorithm 1 uses `SATURATINGCOUNTER` at two places. In lines 2 through 3, we check if the saturated count of solutions of φ is itself within $\text{thresh} - 1$; if so, we return this count directly. Otherwise, we use `SATURATINGCOUNTER` in line 12 to find the saturated count of solutions of φ in cell $\mathbf{0}^i$ induced by h_i for an appropriately chosen $i \in \{1, \dots, n\}$. This count is then recorded in the i^{th} element of an array named `CellSize`. The sub-routine `INITIALIZEARRAY` is used to initialize `CellSize` prior to starting the iterations of the loop in lines 10–15. Specifically, `INITIALIZEARRAY` sets `CellSize[0]` to \top (since the saturated model count of φ is \top if the check in line 3 fails) and `CellSize[i]` to \perp , representing a value that has not been assigned yet, for $1 \leq i \leq n$.

Recall that the count of solutions of φ in a cell induced by h_i is an unbiased estimator of $\frac{|\text{Sol}(\varphi)|}{2^i}$. Therefore, this count reduces (in expectation) as i increases. In order to find the smallest i where this drops below thresh , we check in line 13 if the count induced by h_i is less than thresh , whereas that induced by h_{i-1} is thresh or more (i.e. saturated at \top). If so, we identify i as the smallest index for which the partitioning of $\text{Sol}(\varphi)$ by h_i yields “small” cells. We also record the value of `CellSize[i] $\times 2^i$` as the current estimate of $|\text{Sol}(\varphi)|$ in a list of estimates named `CountEst`. If, however, the check in line 13 fails, we conclude that the cells induced by h_i are either too small (`CellSize[$i - 1$] $\neq \top$`) or too large (`CellSize[i] = \top`). In this case, we continue the search by choosing another index $i \in \{1, \dots, n\}$ such that `CellSize[i]` has not been assigned yet, i.e. has the value \perp . Different hash-based model counters use different strategies for choosing the next unassigned index. This is represented by an invocation of sub-routine `CHOOSEUNASSIGNEDINDEX`

in line 11 of Algorithm 1.

While the loop in lines 10–15 of Algorithm 1 provides an estimate of $|\text{Sol}(\varphi)|$ within a specified tolerance, the confidence may not be as high as required. The loop in lines 7–15 followed by computation of the median of all estimates in the list `CountEst` serves to amplify the probability of success to the desired level. Every iteration of this loop is required to be independent of all previous and subsequent iterations. The number of iterations needed to achieve the desired confidence is a linear function of $\log \frac{1}{\delta}$ and is calculated in line 4.

In the following subsections, we discuss how different choices of \mathcal{H} and of the subroutines used in Algorithm 1 affect the performance and guarantees of various probabilistic hash-based counting algorithms. We start with a brief overview of the theoretical tools and techniques necessary to analyze different choices.

26.2.2. Theoretical tools and analysis

Let $\text{Cell}_{\langle\varphi, h_i\rangle}$ denote the random set $\{y \in \text{Sol}(\varphi) \mid h_i(y) = \mathbf{0}^i\}$. Now consider an arbitrary iteration of the loop in lines 7–15 of Algorithm 1. We say that the iteration *fails* if either no estimate of the model count is appended to `CountEst` in line 14, or if the estimate that is appended does not lie in the interval $\left[\frac{|\text{Sol}(\varphi)|}{1+\varepsilon}, |\text{Sol}(\varphi)| \cdot (1+\varepsilon)\right]$. Let `IterError` denote the event that the iteration fails in the above sense. Furthermore, let T_i denote the event $(|\text{Cell}_{\langle\varphi, h_i\rangle}| < \text{thresh})$, and let L_i and U_i denote the events $\left(|\text{Cell}_{\langle\varphi, h_i\rangle}| \times 2^i < \frac{|\text{Sol}(\varphi)|}{(1+\varepsilon)}\right)$ and $(|\text{Cell}_{\langle\varphi, h_i\rangle}| > |\text{Sol}(\varphi)|(1+\varepsilon))$, respectively. It now follows from the definition of `IterError` that $\Pr[\text{IterError}] = \Pr\left[\bigcap_{j=0}^n \overline{T}_j \cup \bigcup_{i=1}^n (\overline{T}_{i-1} \cap T_i \cap (L_i \cup U_i))\right]$. A key component of the analysis of Algorithm 1 concerns bounding $\Pr[\text{IterError}]$ from above by a constant strictly less than $\frac{1}{2}$. Once this bound is established, standard arguments for probability amplification via median of independent random variables can be used to bound $\Pr[\text{Error}]$, where `Error` denotes the event that Algorithm 1 doesn't yield a count within the prescribed tolerance bound ε .

To see how probability amplification happens, suppose the loop in lines 7–15 is iterated r independent times. Since we compute the median in line 16, the event `Error` can happen only if `IterError` happens for at least $(r/2) + 1$ iterations, where we have assumed r to be even for simplicity. For notational simplicity, let $\rho (< \frac{1}{2})$ denote $\Pr[\text{IterError}]$. Then, $\Pr[\text{Error}] = \sum_{i=(r/2)+1}^r \binom{r}{i} \rho^i (1-\rho)^{r-i}$. Since $\binom{r}{i} \leq \binom{r}{r/2} \leq 2^r$, algebraic simplification yields $\Pr[\text{Error}] < \left(\frac{1-\rho}{1-2\rho}\right) \times \left(2\sqrt{\rho(1-\rho)}\right)^r$. Therefore, in order to ensure $\Pr[\text{Error}] \leq \delta$, we need $r \geq C(\rho) \times \left(\log \frac{1}{\delta} + D(\rho)\right)$, where $C(\rho)$ and $D(\rho)$ depend only on ρ . This is what sub-routine `FINDREPETITIONCOUNT` computes in line 4 of Algorithm 1.

To obtain a good upper bound of ρ , i.e. $\Pr[\text{IterError}]$, we need to find good bounds of probabilities of the events T_i, L_i , and U_i . In general, different techniques are used to compute these bounds, depending on the choice of \mathcal{H} , and of sub-routines `CHOOSEHASHFAMILY` and `CHOOSEHASHFUNCTIONS`. The following three standard probability bounds are particularly useful in reasoning about $|\text{Cell}_{\langle\varphi, h_i\rangle}|$, where $0 < \beta < 1$, and μ_i and σ_i denote the mean and standard

deviation respectively, of $|\text{Cell}_{\langle\varphi, h_i\rangle}|$.

- (Chebyshev) $\Pr [| \text{Cell}_{\langle\varphi, h_i\rangle}| - \mu_i | \geq \beta\mu_i] \leq \frac{(1+\beta)^2\sigma_i^2}{\beta^2\mu_i^2}$
- (Payley-Zygmund) $\Pr [|\text{Cell}_{\langle\varphi, h_i\rangle}| \leq \beta\mu_i] \leq \frac{\sigma_i^2}{\sigma_i^2 + (1-\beta)^2\mu_i^2}$
- (Markov) $\Pr [|\text{Cell}_{\langle\varphi, h_i\rangle}| \geq \frac{\mu_i}{\beta}] \leq \beta$

The recipe for obtaining good upper bounds of the probabilities of T_i , L_i , and U_i is to use appropriate values of β in the above inequalities, and to use hash-family specific properties to obtain expressions for σ_i^2 and μ_i . Existing techniques typically rewrite the expression for $\Pr [\text{IterError}]$ so that the above bounds can be used for $\mu_i > 1$. Significantly, the choice of **thresh** informs the need for bounding either the coefficient of variation, i.e., $\frac{\sigma_i^2}{\mu_i^2}$, or the dispersion index, $\frac{\sigma_i^2}{\mu_i}$, as discussed below.

26.2.3. Choice of threshold

For an algorithm focused on a problem as hard as (approximately) counting the number of solutions of a CNF formula, it is pleasing to note that except the **SATURATINGCOUNTER** query in line 12, Algorithm 1 is fairly simple to implement and execute. Therefore, it is no surprise that practical implementations of the algorithm spend most of their time in **SATURATINGCOUNTER** queries in line 12 [SM19]. A straightforward way of implementing **SATURATINGCOUNTER** with modern SAT solvers is to enumerate solutions one by one, using blocking clauses for already found solutions, until either **thresh** solutions are generated or there are no more solutions. A more generic approach to implementing **SATURATINGCOUNTER** is described in Section 26.2.6. In all of these approaches, the number of times an underlying SAT solver has to be invoked is linear in **thresh**. Therefore, it is important to ensure that **thresh** grows polynomially in n and $\frac{1}{\varepsilon}$.

There is a fundamental tradeoff between **thresh** and the tolerance bound ε that every approximate counter obtained as an instance of Algorithm 1 must respect. To understand this, suppose a hypothetical algorithm **M** computes a multiplicative $(1 + \varepsilon)$ -approximation of a model count that lies in the interval $[1, 2^n]$. The minimum cardinality of the set of possible output values of **M** has an inverse logarithmic relation with $(1 + \varepsilon)$. Specifically, in order to minimize the count of output values of **M**, we must divide the interval $[1, 2^n]$ into non-overlapping sub-intervals such that the ratio of the upper bound to the lower bound of each sub-interval is $(1 + \varepsilon)^2$. Since the model count being estimated can be anywhere in $[1, 2^n]$, algorithm **M** must have the ability to output at least one value from each of the above sub-intervals. If t denotes the number of sub-intervals, then we must have $\frac{2^n}{(1+\varepsilon)^t} < 1$, i.e. $t > \frac{n}{2\log_2(1+\varepsilon)}$. This gives the minimum count of possible estimates that algorithm **M** must return. Notice that the estimated count appended to **CountEst** in the loop in lines 10–15 of Algorithm 1 can be viewed as $c \times 2^i$ for some $1 \leq c < \text{thresh}$ and $i \in [0, n]$. Therefore, the cardinality of the set of possible values returned by Algorithm 1 is $n \times \text{thresh}$. It follows that $n \times \text{thresh} \geq \frac{n}{2\log_2(1+\varepsilon)}$, i.e., $\text{thresh} \geq \frac{1}{2\log_2(1+\varepsilon)}$.

The lower bound of `thresh` derived above may not be achievable in a given instantiation of Algorithm 1, and typically larger values of `thresh` are used. For example, Chakraborty et al [CMV13b, CMV16] have used `thresh` in $\mathcal{O}(1/\varepsilon^2)$ to ensure a multiplicative $(1 + \varepsilon)$ -approximation. Ermon et al. [EGSS14], Asteris and Dimakis [AD16], and Achlioptas and Theodoropoulos [AT17] have, however, used `thresh` in $\mathcal{O}(1/\varepsilon)$ for the same purpose. Not surprisingly, the choice of `thresh` affects the choice of hash functions that can be used in Algorithm HASHCOUNTER. Specifically, with `thresh` $\in \mathcal{O}(1/\varepsilon^2)$, the dispersion index of $|\text{Cell}_{\langle \varphi, h_i \rangle}|$, i.e. $\frac{\sigma_i^2}{\mu_i}$, must be bounded above by a constant. In contrast, when `thresh` $\in \mathcal{O}(1/\varepsilon)$, the coefficient of variation of $|\text{Cell}_{\langle \varphi, h_i \rangle}|$, i.e. $\frac{\sigma_i^2}{\mu_i^2}$, must be bounded by a constant.

26.2.3.1. Multiplicative approximation amplification

The above discussion implies that as ε decreases, `thresh` must increase if we aspire to obtain a multiplicative $(1 + \varepsilon)$ -factor approximation. If `thresh` becomes too large, the running time of SATURATINGCOUNTER may become a bottleneck. It is therefore interesting to ask if we can avoid increasing `thresh` at the cost of increasing the size of the formula fed as input to SATURATINGCOUNTER, while still obtaining a $(1 + \varepsilon)$ -factor approximation. In 1983, Stockmeyer [Sto83] observed that this can indeed be done: one can transform an algorithm that provides a constant-factor approximation into one that gives a multiplicative $(1 + \varepsilon)$ -approximation. To see how this can be achieved, suppose an algorithm **M** computes model counts with a 2-approximation. Given an input formula φ over the set of variables X , suppose we make two copies of the formula with disjoint sets of variables, i.e. $\psi(X, Y) = \varphi(X) \wedge \varphi(Y)$ where $\varphi(Y)$ is the same formula as $\varphi(X)$, but with variables X replaced by a new set of variables Y . It is easy to see that $|\text{Sol}(\psi)| = |\text{Sol}(\varphi)|^2$. Furthermore, if $\frac{|\text{Sol}(\psi)|}{4} \leq c \leq 4|\text{Sol}(\psi)|$, then $\frac{|\text{Sol}(\varphi)|}{2} \leq \sqrt{c} \leq 2|\text{Sol}(\varphi)|$. Thus, we have effectively reduced the constant multiplicative factor in approximating $|\text{Sol}(\varphi)|$. Extending this argument, if we want an $(1 + \varepsilon)$ -approximation of $|\text{Sol}(\varphi)|$ for small values of ε , we can construct ψ by conjoining $k = \mathcal{O}(1/\varepsilon)$ copies of φ , each with a disjoint set of variables, and compute the k -th root of the estimate of $|\text{Sol}(\psi)|$ returned by algorithm **M**. Therefore, `thresh` can be set to a constant when computing an $(1 + \varepsilon)$ -approximation of $|\text{Sol}(\varphi)|$ if we are willing to invoke Algorithm 1 over a new formula obtained by conjoining $\mathcal{O}(1/\varepsilon)$ copies of φ .

The need for multiple copies of the given formula, however, leads to larger queries, to which SAT solvers are highly sensitive. For classes of formulas not closed under conjunction, the query $\text{SATURATINGCOUNTER}(\psi \wedge h_i^{-1}(\mathbf{0}^i), \text{thresh})$ may not even lie in the same complexity class as $\text{SATURATINGCOUNTER}(\varphi \wedge h_i^{-1}(\mathbf{0}^i), \text{thresh})$. As an example, the class of DNF formulas is not closed under conjunction. If φ is a DNF formula and ψ is a conjunction of DNF formulas, the query $\text{SATURATINGCOUNTER}(\varphi \wedge h_i^{-1}(\mathbf{0}^i), \text{thresh})$ can be answered in polynomial time, while the query $\text{SATURATINGCOUNTER}(\psi \wedge h_i^{-1}(\mathbf{0}^i), \text{thresh})$ is NP-hard in general. This is a crucial consideration for the design of hashing-based FPRAS techniques, which are discussed in detail in Section 26.4.

26.2.4. Choice of hash family

Extending the notation introduced earlier, let $h_i \stackrel{R}{\leftarrow} \mathcal{H}_i$ denote the probability space induced by choosing a function $h_i : \{0, 1\}^n \rightarrow \{0, 1\}^i$ uniformly at random from the sub-family \mathcal{H}_i .

Definition 26.2.1. A sub-family of hash functions \mathcal{H}_i is said to be *uniform* if for all $x \in \{0, 1\}^n$, $\alpha \in \{0, 1\}^i$ and $h_i \stackrel{R}{\leftarrow} \mathcal{H}_i$, $\Pr[h_i(x) = \alpha] = \frac{1}{2^i}$.

Definition 26.2.2. [CW77] A sub-family of hash functions \mathcal{H}_i is said to be *strongly 2-universal* if for all $x, y \in \{0, 1\}^n$ and $h_i \stackrel{R}{\leftarrow} \mathcal{H}_i$, $\Pr[h_i(x) = h_i(y)] = \frac{1}{2^i}$.

Uniform and strongly 2-universal hash functions play a central role in the design and analysis of probabilistic hash-based approximate counters.

Early work on probabilistic hash-based counting, viz. [GSS06, GHSS07b, GHSS07a], made the important observation that properties of strongly 2-universal hash families allow one to design approximate counters with PAC guarantees. Specifically, using strongly 2-universal hash families ensures that the dispersion index of $|\text{Cell}_{\langle F, h_i \rangle}|$, i.e., $\frac{\sigma_i^2}{\mu_i}$, is bounded above by a constant. Note that if $\mu_i > 1$, this also implies that the coefficient of variation is bounded above by a constant. Furthermore, it was observed that there exist strongly 2-universal hash families for which the problem of finding a $y \in h_i^{-1}(\mathbf{0}^i)$ is in NP, and hence can be solved using SAT solvers. Among various hash families that have been studied over the years, two have featured predominately in the literature on hashing-based model counting. These are the families of random XOR-based hash functions, denoted \mathcal{H}_{xor} , and Toeplitz matrix [Gra06] based hash functions, denoted $\mathcal{H}_{\mathcal{T}}$. If we view the variables x_1, x_2, \dots, x_n in the support of the formula φ as a vector X of dimension $n \times 1$, a hash function $h_i : \{0, 1\}^n \mapsto \{0, 1\}^i$ chosen from either \mathcal{H}_{xor} or $\mathcal{H}_{\mathcal{T}}$ can be represented as $h_i(X) = \mathbf{A}X + \mathbf{b}$, where \mathbf{A} is an $m \times n$ 0-1 matrix, \mathbf{b} is $m \times 1$ 0-1 vector and all operations are in \mathbf{GF}_2 . Regardless of whether we use \mathcal{H}_{xor} or $\mathcal{H}_{\mathcal{T}}$, the vector \mathbf{b} is chosen uniformly at random from the space of all $m \times 1$ 0-1 vectors. The way in which \mathbf{A} is chosen, however, differs depending on whether we are using the family \mathcal{H}_{xor} or $\mathcal{H}_{\mathcal{T}}$. In case we are using \mathcal{H}_{xor} , \mathbf{A} is chosen uniformly at random from the space of all $m \times n$ 0-1 matrices. If, on the other hand, we are using $\mathcal{H}_{\mathcal{T}}$, \mathbf{A} is chosen uniformly at random from the space of all $m \times n$ Toeplitz 0-1 matrices. Note that Toeplitz 0-1 matrices form a very small subset of all 0-1 matrices. While no significant performance difference has been reported for probabilistic hash-based approximate counters using these two families of hash functions, the family \mathcal{H}_{xor} seems to be the hash family of choice in the existing literature.

When using the family \mathcal{H}_{xor} , the random selection of h_i can be achieved by choosing each entry of \mathbf{A} to be 1 with probability $p = 1/2$. On an average, this gives $\frac{n}{2}$ 1's in each row of \mathbf{A} . The invocation of SATURATINGCOUNTER in Algorithm 1 necessitates usage of a SAT solver to solve the formula $\varphi \wedge (\mathbf{A}X + \mathbf{b} = 0)$. Thus, the SAT solver needs to reason about formulas that are presented as a conjunction of usual (i.e., OR) clauses and XOR clauses, and each XOR clause has an average size of $\frac{n}{2}$. Gomes et al. [GHSS07b] have observed that the performance of SAT solvers, however, degrades with an increase in the size of XOR clauses.

Therefore recent efforts have focused on the design of sparse hash functions where the count of 1's in every row is $\ll \frac{n}{2}$ [EGSS14, IMMV15, AD16, AT17, AHT18].

Building on the classical notion of definability due to Beth [Bet56], one can define the notion of an *independent support* of a Boolean formula. Specifically, $\mathcal{I} \subseteq X$ is an independent support of φ if whenever two solutions π_1 and π_2 of φ agree on \mathcal{I} , then $\pi_1 = \pi_2$ [LM08, CMV14, LLM16]. Chakraborty et al. [CMV14] observed that strongly 2-universal hash functions defined only over \mathcal{I} (instead of X) lend themselves to exactly the same reasoning as strongly 2-universal hash functions defined over the whole of X , when used to partition the solution space of φ . The importance of this observation comes from the fact that for many important classes of problems, the size of \mathcal{I} can be one to two orders of magnitude smaller than that of X . This in turn leads to XOR clauses that are one to two orders of magnitude smaller than that obtained by using strongly 2-universal hash functions defined over X [IMMV15]. Ivrii et al. have proposed an algorithmic technique, called MIS, to determine the independent support of a given formula via a reduction to the Group-oriented Minimal Unsatisfiable Subformula (GMUS) problem [LS08, Nad10]. Given the hardness of GMUS, the proposed technique MIS can scale to a few tens of thousands of variables, and designing scalable techniques for GMUS is an active area of research.

Gomes et al. [GHSS07b] observed that for some classes of formulas, setting the expected fraction p of 1's in each row of the matrix \mathbf{A} to be significantly smaller than $1/2$ can still provide counts close to those obtained using $p = 1/2$. A straightforward calculation shows that if we choose $h_i(X) = \mathbf{A}X + \mathbf{b}$, with every entry in \mathbf{A} set to 1 with probability p , then for any two distinct vectors $x, y \in \{0, 1\}^n$, $\Pr[h_i(x) = h_i(y)] = (1/2 + 1/2(1-2p)^w)^i = q(w)$ where w is the Hamming weight of $x - y$, i.e., the number of ones in $x - y$. Note that for $p < 1/2$, such a family is not strongly 2-universal. Ermon et al. [EGSS14] observed, however, that using a strongly 2-universal hash family is not a necessary requirement to bound the coefficient of variation from above by a constant. Furthermore, for a given $x \in \{0, 1\}^n$ and for any $k \in \{1, \dots, n\}$, the count of $y \in \{0, 1\}^n$ such that $\Pr[h(x) = h(y)] = q(k)$ is bounded from above by $\binom{n}{k}$. This allowed Ermon et al. to analytically compute values of p that are less than $\frac{1}{2}$, although obtaining a closed-form expression remained elusive. The gap between analytical bounds and closed form bounds was bridged by Zhao et al. [ZCSE16] and Asteris and Dimakis [AD16], who showed that having $p \in \mathcal{O}(\frac{\log i}{i})$ suffices to upper bound the coefficient of variation when $i \in \Theta(n)$. Observe that the smaller the value of p , the sparser the matrix \mathbf{A} .

Motivated by the use of sparse matrices in the design of efficient error-correcting codes, Achlioptas and Theodoropoulos [AT17] showed that one could randomly choose \mathbf{A} from an ensemble of Low-Density Parity Check (LDPC) matrices. Achlioptas, Hammoudeh, and Theodoropoulos [AHT18] put forth the observation that an approximate counter strives to compute an estimate c that satisfies two bounds: (i) a lower bound: $c \leq |\text{Sol}(\varphi)| \cdot (1 + \varepsilon)$ and (ii) an upper bound: $c \geq \frac{\text{Sol}(\varphi)}{1 + \varepsilon}$. Achlioptas et al. [AT17, AHT18] and Gomes et al. [GHSS07b] further observed that obtaining good lower bounds requires the hash family \mathcal{H} to be uniform, while obtaining good upper bounds requires \mathcal{H} to ensure that the coefficient of variation of $|\text{Cell}_{\langle \varphi, h_i \rangle}|$ is bounded from above by a constant.

There are a few issues that arise when translating the above ideas to a hash-based approximate counter with PAC guarantees that also scales in practice. First, the bounds achieved with $p \in \mathcal{O}(\frac{\log i}{i})$ hold only for $i \in \Theta(n)$, while for most practical applications $i \ll n$. Second, bounding the coefficient of variation suffices to provide constant-factor approximation guarantees. In order to guarantee $(1 + \varepsilon)$ -approximation, Stockmeyer’s technique [Sto83] of reducing the tolerance must be used, which in turn leads to large formulas being fed to the SAT solver underlying SATURATINGCOUNTER.

In a departure from earlier works [EGSS14, AD16, ZCSE16] where the focus was to use analytical methods to obtain upper bound on the variance of $|\text{Cell}_{\langle \varphi, h_i \rangle}|$, Meel $\text{\textcircled{R}}$ Akshay [MA20] focused on searching for the set $\text{Sol}(F)$ that would achieve the maximum variance of $|\text{Cell}_{\langle \varphi, h_i \rangle}|$. This allowed them to observe an elegant connection between the maximization of variance as well as dispersion index of $|\text{Cell}_{\langle \varphi, h_i \rangle}|$ and minimization of the “ t -boundary” (the number of pairs with Hamming distance at most t) of sets of points on the Boolean hypercube on n dimensions. Utilizing this connection, they introduced a new family of hash functions, denoted by $\mathcal{H}_{\text{Rennes}}$, which consists of hash functions of the form $\mathbf{A}y + b$, where every entry of $\mathbf{A}[i]$ is set to 1 with $p_i = \mathcal{O}(\frac{\log_2 i}{i})$. The construction of the new family marks a significant departure from prior families in that the density of 1’s in the matrix \mathbf{A} is dependent on the row index of the matrix. Meel $\text{\textcircled{R}}$ Akshay demonstrated that usage of $\mathcal{H}_{\text{Rennes}}$ can lead to significant speedup in runtime while preserving PAC guarantees [MA20].

26.2.5. Using dependent hash functions

Recall that in line 8 of Algorithm 1, the vector of hash functions (h_0, \dots, h_n) is chosen such that h_i maps $\{0, 1\}^n$ to $\{0, 1\}^i$. As already discussed in Section 26.2.2, an analysis and proof of correctness of an instantiation of Algorithm 1 requires us to bound $\text{Pr}[\text{IterError}]$, which in turn involves analyzing expressions involving L_i, U_i , and T_i for different values of i . Since, the dependence of L_i, U_i , and T_i across different values of i stems from the choice of (h_0, \dots, h_n) , a natural alternative is to choose each h_i uniformly randomly from the sub-family \mathcal{H}_i , but independently of the choice of h_j for all $j \neq i$. This minimizes the dependence among events that correspond to different values of i . Not surprisingly, early work on probabilistic hash-based counting relied on this independence of the chosen hash functions for theoretical analysis. Interestingly, independence in the choice of h_i ’s necessitates a linear search among the indices to find the smallest i that satisfies the check in line 13. This is because with independently chosen h_i ’s, one cannot rule out the possibility that $\text{CellSize}[i - 1] < \text{thresh}$ but $\text{CellSize}[i] = \top$. Independence in the choice of h_i ’s, therefore, requires that sub-routine CHOOSEUNASSIGNEDINDEX chooses indices in sequential order from 1 through n . Indeed, it is not known whether we can do better than linear search when h_i ’s are chosen independently.

At this point, one may ask if there is anything to be gained from dependence among hash functions. In this context, the *prefix hash family*, used in [CMV16], is worth examining. Let $\alpha \in \{0, 1\}^i$ be a vector. For $1 \leq j \leq i$, we use $\alpha[j]$ to denote the j^{th} component of α , and $\alpha[1..j]$ to denote the projection of α on the first j dimensions, i.e. $(\alpha[1], \dots, \alpha[j])$. For every hash function $h_i : \{0, 1\}^n \rightarrow \{0, 1\}^i$, we

now define the j^{th} *prefix-slice* of h_i , denoted $h_i[1..j]$ as a map from $\{0, 1\}^n$ to $\{0, 1\}^j$ such that $h_i[1..j](x) = h_i(x)[1..j]$, for all $x \in \{0, 1\}^n$.

Definition 26.2.3. [CMV16] A family of hash functions $\{\mathcal{H}_1, \dots, \mathcal{H}_n\}$ is called a *prefix-family* if for all $h_i \in \mathcal{H}_i$, the j^{th} prefix-slice of h_i is in \mathcal{H}_j for all $j \in \{1, \dots, i\}$. Thus, for all $h_i \in \mathcal{H}_i$, there exists $h_{i-1} \in \mathcal{H}_{i-1}$ such that for every $x \in \{0, 1\}^n$, $h_i(x)$ is simply $h_{i-1}(x)$ augmented with the i^{th} component of $h_i(x)$.

Suppose the hash functions (h_0, \dots, h_n) are chosen from a prefix family such that $h_i = h_n[1..i]$ for $1 \leq i \leq n$ ($h_0 : \{0, 1\}^n \rightarrow \{0\}$ leaves no choice in design). It is easy to see that with this choice of hash functions, $|\text{Cell}_{\langle \varphi, h_i \rangle}| \leq |\text{Cell}_{\langle \varphi, h_{i-1} \rangle}|$ for all $i \in \{1, \dots, n\}$. Therefore, one can perform a binary search to find the smallest (and unique) i such that $\text{CellSize}[i] < \text{thresh}$ but $\text{CellSize}[i-1] = \top$. The runtime of known implementations of $\text{SATURATINGCOUNTER}(\varphi \wedge h_i^{-1}(\mathbf{0}^i))$ has, however, been empirically observed to significantly increase with i . Therefore, a galloping search performs significantly better than binary search. Furthermore, notice that if C is the set of clauses learnt (as by a CDCL SAT solver) while solving $\varphi \wedge h_i^{-1}(\mathbf{0}^i)$, the dependence among different hash functions facilitates incremental SAT solving. In other words, the clauses learnt while solving $(\varphi \wedge h_i^{-1}(\mathbf{0}^i))$ are sound (and hence, can be re-used) for solving $(\varphi \wedge h_j^{-1}(\mathbf{0}^j))$ for $j > i$. While this sounds promising, one needs to re-think theoretical proofs of PAC guarantees, when using such dependent hash functions. Indeed, the dependence among different events may force usage of union bounds over linear (in n) terms when computing $\Pr[\text{IterError}]$, leading to overall weaker bounds. By a carefully constructed sequence of arguments, Chakraborty et al. showed in [CMV16] that one can indeed use dependence to one's advantage even for the theoretical analysis of PAC guarantees. In particular, they showed that when h_i 's are chosen from a prefix family as described above, $\Pr[\text{IterError}] \leq 0.36$. The work of Achlioptas and Theodoropoulos [AT17, AHT18], in which LDPC matrices were used to define hash functions, also provides another example where dependence between h_i 's have been used to advantage in designing better hash-based approximate model counters.

26.2.6. Counting up to a threshold

Finally, we discuss the key sub-routine SATURATINGCOUNTER used in Algorithm 1. Recall that SATURATINGCOUNTER is used to find the saturated count (upto thresh) of models of $\varphi \wedge h_i^{-1}(\mathbf{0}^i)$. Computationally, this is the most expensive step of Algorithm 1. We now show that if satisfiability checking of $\varphi \wedge h_i^{-1}(\mathbf{0}^i)$ is *self-reducible*, then SATURATINGCOUNTER can be implemented with at most $\mathcal{O}(n \cdot \text{thresh})$ calls to a satisfiability checker for $\varphi \wedge h_i^{-1}(\mathbf{0}^i)$. Fortunately, satisfiability checking is indeed self-reducible for a large class of formulas φ and hash functions h_i .

While a general definition of self-reducibility can be found in [Bal88], our requirement can be stated simply as follows. Let \mathcal{C} be a class of propositional formulas, viz. CNF, DNF, CNF+XOR, DNF+XOR, etc. and let ψ be a formula in \mathcal{C} . As an example, if φ is in CNF and $h_i^{-1}(\mathbf{0}^i)$ is a conjunction of XOR clauses, then $\varphi \wedge h_i^{-1}(\mathbf{0}^i)$ is a formula in the CNF+XOR class. Let x_1, \dots, x_n be the

variables in ψ and let $(\pi_1, \dots, \pi_n) \in \{0, 1\}^n$ be an assignment of these variables. Let $\psi|_{x_1 \dots x_i = \pi_1 \dots \pi_i}$ denote the formula obtained by substituting π_j for x_j in ψ , for all $j \in \{1, \dots, i\}$. We say that satisfiability checking is *self-reducible* for the class \mathcal{C} if for every formula $\psi \in \mathcal{C}$, the following hold: (i) $\psi|_{x_1 = \pi_1}$ is in \mathcal{C} , and (ii) $(\pi_1, \dots, \pi_n) \models \psi \Leftrightarrow (\pi_2, \dots, \pi_n) \models \psi|_{x_1 = \pi_1}$. Continuing with our earlier example, let \mathcal{C} denote the CNF+XOR class and ψ denote the formula $\varphi \wedge h_i^{-1}(\mathbf{0}^i)$ in class \mathcal{C} . It is easy to see that (i) $\psi|_{x_1 = \pi_1}$ continues to be a CNF+XOR formula, and (ii) $(\pi_1, \dots, \pi_n) \models \psi \Leftrightarrow (\pi_2, \dots, \pi_n) \models \psi|_{x_1 = \pi_1}$. Hence, satisfiability checking is self-reducible for the CNF+XOR class.

Suppose we have access to a satisfiability checker for formulas in \mathcal{C} . If ψ is unsatisfiable, a single invocation of the checker suffices to detect it. Otherwise, using a standard self-reducibility argument, a solution of ψ can be obtained using at most $\mathcal{O}(n)$ calls to the satisfiability checker. Let $\pi = (\pi_1, \dots, \pi_n)$ be such a solution. For $1 \leq i \leq n$, let $\tilde{\pi}_i$ denote $(\pi_1, \dots, \pi_{i-1}, \bar{\pi}_i)$, where $\bar{1} = 0$ and $\bar{0} = 1$. Following a technique used by Murty [Mur68], we can now partition $\text{Sol}(\psi)$ into $n + 1$ (possibly empty) sets Z_0, \dots, Z_n , where $Z_0 = \{\pi\}$ and $Z_i = \{\tilde{\pi}_i \tau \mid \tau \in \text{Sol}(\psi|_{x_1 \dots x_i = \tilde{\pi}_i})\}$ for $1 \leq i \leq n - 1$, with $\tilde{\pi}_i \tau$ denoting the assignment obtained by concatenating $\tilde{\pi}_i$ and τ . Finally, $Z_n = \{\tilde{\pi}_n\}$ if $\tilde{\pi}_n \models \psi$ and $Z_n = \emptyset$ otherwise. As an example, if $n = 3$ and $\pi = (0, 1, 1)$, then $\text{Sol}(\psi)$ is partitioned into four sets: (i) $Z_0 = \{(0, 1, 1)\}$, (ii) $Z_1 = \{(1, \tau_1, \tau_2) \mid (\tau_1, \tau_2) \in \text{Sol}(\psi|_{x_1=1})\}$, (iii) $Z_2 = \{(0, 0, \tau'_1) \mid \tau'_1 \in \text{Sol}(\psi|_{x_1, x_2=0,0})\}$, and (iv) $Z_3 = \{(0, 1, 0)\}$ assuming $(0, 1, 0) \models \psi$. Observe that starting with a formula ψ with n variables, the above step yields one solution (viz., π) and n self-reduced problem instances (viz. $\psi|_{x_1 \dots x_i = \tilde{\pi}_i}$ for $1 \leq i \leq n$), each with strictly fewer than n variables in their support.

In order to implement SATURATINGCOUNTER, i.e. count the solutions of ψ upto **thresh**, we apply the same step as above to each of the self-reduced problem instances. We stop when a total of **thresh** solutions of ψ have been generated or when all remaining self-reduced problem instances are unsatisfiable. Observe that whenever a new satisfying assignment is found, $\mathcal{O}(n)$ self-reduced problem instances are generated. Since at most **thresh** solutions of ψ are generated (and counted) by SATURATINGCOUNTER, only $\mathcal{O}(n \cdot \text{thresh})$ self-reduced problem instances can be generated. For each of these, we either detect unsatisfiability using a single satisfiability check or obtain a solution using $\mathcal{O}(n)$ satisfiability checks. The latter can, however, happen at most **thresh** times. Therefore, the total number of calls to the satisfiability checker is in $\mathcal{O}(n \cdot \text{thresh})$.

The above technique of implementing SATURATINGCOUNTER using at most $\mathcal{O}(n \cdot \text{thresh})$ invocations of a satisfiability checker works for all class of problems for which satisfiability checking is self-reducible. This includes cases like DNF+XOR formulas, perfect matchings in bi-partite graphs etc.

If \mathcal{C} is a class of propositional formulas for which satisfiability checking is NP-complete, then checking satisfiability of $\varphi \wedge h_i^{-1}(\mathbf{0}^i)$, where $\varphi \in \mathcal{C}$, is also NP-complete. In such cases, we are often interested in the query complexity of SATURATINGCOUNTER, given access to an NP oracle. Interestingly, the technique of implementing SATURATINGCOUNTER as discussed above may not yield optimal query complexity in such cases. Observe that the above technique not only finds the saturated count of solutions but also generates the corresponding

set of solutions. Bellare, Goldreich, and Petrank [BGP00] observed that if we are interested only in the saturated count, then SATURATINGCOUNTER can be implemented with $\mathcal{O}(\text{thresh})$ queries to an NP oracle. Bellare et al.'s technique, however, constructs queries of size $\mathcal{O}(|\varphi| \cdot \text{thresh})$. While this is a polynomial blow-up in the size of the formula fed to the oracle, the performance of modern SAT solvers can be significantly impacted by such a blow up.

It is worth noting here that while an NP oracle simply provides a Yes/No answer to a decision query, modern SAT solvers either provide a satisfying assignment for an input formula or report unsatisfiability of the formula. Given this difference, it is interesting to ask how many invocations of a SAT solver would be needed in order to implement SATURATINGCOUNTER. To formalize this question, we introduce the notion of a *SAT oracle* that takes a propositional formula φ as input, and returns a solution σ if φ is satisfiable and \perp if φ is unsatisfiable. We can now implement SATURATINGCOUNTER by simply enumerating the satisfying assignments of an input formula φ using a SAT oracle, until the oracle either returns \perp or we have already found thresh solutions. It is not hard to see that this requires $\mathcal{O}(\text{thresh})$ invocations of the SAT oracle, and the largest size of a formula used in any invocation is $\mathcal{O}(|\varphi| + \text{thresh} \cdot |\text{Sup}(\varphi)|)$.

ApproxMC: A specific hash-based approximate counter

As discussed earlier, state-of-the-art probabilistic hash-based approximate counters (for CNF formulas) can be viewed as (optimized) instances of the generic Algorithm 1. One such counter is ApproxMC [CMV16], obtained with the following specific choices in Algorithm 1.

- In line 1 of the algorithm, thresh is set to $\mathcal{O}(1/\varepsilon^2)$.
- In line 4, itercount is set to $\mathcal{O}(\log_2 \delta^{-1})$.
- In line 5, the prefix hash family \mathcal{H}_{xor} is chosen.
- In line 8, individual hash functions are chosen such $h_i = h_n[1..i]$ for $1 \leq i \leq n$, with $h_0 : \{0, 1\}^n \rightarrow \{0\}$.
- SATURATINGCOUNTER(φ, thresh) is implemented using $\mathcal{O}(\text{thresh})$ invocations of a SAT oracle (a SAT solver, CryptoMiniSat, in practice).

It is shown in [CMV16] that the resulting algorithm provides the following PAC guarantee.

Theorem 26.2.1. Given a formula φ , tolerance $\varepsilon (> 0)$, and confidence parameter δ ($0 \leq \delta < 1$), invocation of ApproxMC($\varphi, \varepsilon, \delta$) returns a count c such that $\Pr[|\text{Sol}(\varphi)| \cdot (1 + \varepsilon)^{-1} \leq c \leq |\text{Sol}(\varphi)|(1 + \varepsilon)] \geq 1 - \delta$. Furthermore, ApproxMC makes $\mathcal{O}(\log n \cdot \varepsilon^{-2} \cdot \log \delta^{-1})$ calls to a SAT oracle.

26.3. Handling CNF+XOR Constraints

Given the extensive reliance on XOR clauses for partitioning the solution space of φ , it is desirable that the underlying SAT solver in SATURATINGCOUNTER have native support for conjunctions of XOR clauses. While a system of only XOR clauses can be solved in polynomial-time via Gauss-Jordan Elimination (GJE),

predominant CDCL SAT solvers are known to have poor performance in their ability to handle XORs [BM00]. Given the importance and usage of XOR clauses in cryptanalysis, there is a rich and long history of building hybrid solvers that perform CDCL and/or lookahead reasoning for CNF clauses and Gauss-Jordan elimination for XOR clauses [BM00, HDVZVM04, Che09, SNC09, Soo10, LJN10, LJN11, LJN12].

Soos, Nohl and Catelluccia [SNC09] proposed an elegant architecture, best viewed as an instance of CDCL(XOR), that keeps CNF and XOR clauses separately. It performs Gauss-Jordan Elimination on XORs while performing CDCL reasoning on CNF constraints, with support for sharing of the learnt clauses from XOR-based reasoning to CNF-based reasoning. This architecture formed the basis of the widely used SAT solver *CryptoMiniSat*. Han and Jiang [HJ12] observed that performing Gauss-Jordan Elimination ensures that the matrix representing XORs is in row echelon form, which allows for lazy and incremental matrix updates. While the separation of CNF and XOR clauses shares similarities to the architecture of modern SMT solvers that separate different theory clauses, there are important differences as well. The primary difference stems from the fact that SMT solvers reason about clauses over different theories with disjoint signatures, while CNF and XOR clauses are defined over the same set of variables. Han and Jiang [HJ12] observed that partial interpolant generation can still be achieved in the presence of mixed CNF and XOR clauses, and the generated interpolants improve the runtime performance of the solver. Laitinen, Junttila, and Niemelä [LJN12] observed that XOR reasoning can be significantly improved by splitting XORs into different *components* that are connected to each other only by *cut variables*. The decomposition ensures that full propagation for each of the components guarantee propagation for the entire set of XORs. Such a decomposition has been empirically shown to improve the memory usage when solving a conjunction of mixed CNF and XOR clauses.

The architecture of separating XOR and CNF clauses, however, comes at the cost of disabling execution of in-processing steps. Upon closer inspection, the separation of CNF and XOR clauses does not fundamentally imply lack of soundness of in-processing steps. Nevertheless, a sound implementation of in-processing steps is a daunting task, given the need for extensive studies into the effect of every in-processing step on XOR clauses. The lack of usage of pre- and in-processing steps significantly hurts the performance of the backend SAT solver in *SATURATINGCOUNTER* since these techniques have been shown to be crucial to the performance of state-of-the-art SAT solvers.

To allow seamless integration of pre- and in-processing steps, it is important that the solver has access to XOR clauses in CNF form while ensuring native support for XORs to perform Gauss-Jordan elimination. To achieve such integration, Soos and Meel [SM19] recently proposed a new architecture called *BIRD* (an acronym for Blast, In-process, Recover, and Destroy). A high-level pseudocode for *BIRD* is shown in Algorithm 2.

Note that Algorithm *BIRD* exits its main loop as soon as it finds a solution or proves that the formula is unsatisfiable. Furthermore, if the benchmarks have XOR clauses encoded in CNF, *BIRD* can efficiently recover such XORs and use Gauss-Jordan elimination on such recovered XORs. The primary challenge for

Algorithm 2 BIRD(φ) $\triangleright \varphi$ has a mix of CNF and XOR clauses

- 1: **Blast** XOR clauses into normal CNF clauses
 - 2: **In-process** (and pre-process) over CNF clauses
 - 3: **Recover** simplified XOR clauses
 - 4: Perform CDCL on CNF clauses with on-the-fly Gauss-Jordan Elimination on XOR clauses until: (a) in-processing is scheduled, (b) a solution is found, or (c) formula is found to be unsatisfiable
 - 5: **Destroy** XOR clauses and goto line 2 if conditions (b) or (c) above don't hold. Otherwise, **return** solution or report unsatisfiable.
-

BIRD is to ensure that line 3 can be executed efficiently. Towards this end, the core idea of the recovery algorithm proposed by Soos and Meel is to choose a clause, say `base_cl`, and recover the unique XOR defined exactly over the variables in `base_cl`, if such an XOR exists. For example, if `base_cl` is $(x_1 \vee x_2 \vee x_3)$, then the only XOR defined over x_1, x_2, x_3 in which `base_cl` can possibly participate is $x_1 \oplus x_2 \oplus x_3 = 1$. Observe that the right hand side of the XOR clause (i.e., 0 or 1) can be obtained by computing the parity of the variables in `base_cl`. The key idea behind the search for XORs over variables in `base_cl`, say of size t , is to perform a linear pass and check whether there exists a subset of CNF clauses that would imply the 2^{t-1} combination of CNF clauses over t variables that corresponds to the desired XOR clause over these t variables.

Note that a clause may imply multiple CNF clauses over t variables. For example, let `base_cl` := $(x_1 \vee x_2 \vee x_3)$, then a clause `cl` := (x_1) would imply 4 clauses over $\{x_1, x_2, x_3\}$, i.e. $\{(x_1 \vee x_2 \vee \neg x_3), (x_1 \vee x_2 \vee x_3), (x_1 \vee \neg x_2 \vee x_3), (x_1 \vee \neg x_2 \vee \neg x_3)\}$. To this end, Soos and Meel maintain an array of possible combinations of size 2^t . They update the entry, indexed by the binary representation of the clause for a fixed ordering of variables, corresponding to a clause `cl'` to 1 if `cl` \rightarrow `cl'`. Similar to other aspects of SAT solving, efficient data structures are vital to perform the above mentioned checks and updates efficiently. The interested reader is referred to [SM19, SGM20] for a detailed discussion.

26.4. Approximate Model Counting for DNF

We now turn our attention to computing a PAC estimate for $|\text{Sol}(\varphi)|$, where φ is a DNF formula. While the problem of exact counting for DNF formulas is #P-complete, Karp and Luby showed in their seminal work [KL83] that there exists an FPRAS for the problem. Specifically, their work used the Monte Carlo framework to arrive at an FPRAS for DNF counting. Subsequently, there have been several follow-up works based on Monte Carlo counting techniques [KLM89, DKLR00, Vaz13, Hub17, HJ19]. These have yielded an improved theoretical understanding of DNF counting and also algorithms that have improved performance in practice. Recently, Chakraborty et al. [CMV16] and Meel, Shrotri, and Vardi [MSV17] showed that the hashing-based framework discussed in Section 26.2.1 could also be adapted to yield an FPRAS for DNF counting. Thus, we have two different approaches for obtaining an FPRAS for DNF counting. These are discussed in detail below.

We fix some notation before delving into the details. A DNF formula φ is

given as a disjunction of cubes. We assume the cubes are indexed by natural numbers, and we denote the i^{th} cube by φ^i . Thus $\varphi = \varphi^1 \vee \dots \vee \varphi^m$. We use n and m to denote the number of variables and cubes, respectively, in the input DNF formula. The *width* of a cube φ^i refers to the number of literals in φ^i and is denoted by $\text{width}(\varphi^i)$. We use w to denote the minimum width, minimized over all cubes of the formula, i.e. $w = \min_i \text{width}(\varphi^i)$.

26.4.1. DNF counting via Monte Carlo framework

Approximate counting algorithms designed using the Monte Carlo approach effectively determine the value of an estimator through multiple independent random samples. The seminal work of Karp and Luby [KL83] can be viewed as a Monte Carlo algorithm using a 0-1 estimator, as shown in Algorithm 3. This algorithm

Algorithm 3 Monte-Carlo-Count(\mathcal{A}, \mathcal{U}) $\triangleright \mathcal{A} \subseteq \mathcal{U}$

```

1:  $Y \leftarrow 0$ 
2: repeat  $N$  times
3:   Select an element  $t \in \mathcal{U}$  uniformly at random
4:   if  $t \in \mathcal{A}$  then
5:      $Y \leftarrow Y + 1$ 
6: until
7:  $Z \leftarrow \frac{Y}{N} \times |\mathcal{U}|$ 
8: return  $Z$ 

```

estimates the cardinality of a set \mathcal{A} in the universe \mathcal{U} , given access to a sub-routine that samples uniformly from \mathcal{U} and a sub-routine that tests if a randomly chosen element is in the set \mathcal{A} . Here, $\frac{Y}{N}$ is an unbiased estimator for $\rho = \frac{|\mathcal{A}|}{|\mathcal{U}|}$ and Z is an unbiased estimator for $|\mathcal{A}|$. It can be shown [R70] that if $N \geq \frac{1}{\rho \varepsilon^2} \cdot \ln(\frac{2}{\delta})$, then $\Pr[(1 - \varepsilon) \cdot |\mathcal{A}| \leq Z \leq (1 + \varepsilon) \cdot |\mathcal{A}|] \geq 1 - \delta$. Algorithm 3 is an FPRAS if the number of samples N , and the times taken by the sampler in line 3 and by the inclusion checker in line 4, are polynomial in the size of the input.

In the context of counting solutions of DNF formulas, the set \mathcal{U} is the set of all assignments over n variables, and $\mathcal{A} = \text{Sol}(\varphi)$. In this case, the ratio ρ is also called the *density of solutions*. A trivial lower bound on $|\text{Sol}(\varphi)|$ is 2^{n-w} . Thus, $\frac{1}{\rho} \leq 2^w$. If w is bounded above by a logarithmic function of n and m , then $\frac{1}{\rho}$ is polynomial in n and m and we need polynomially many samples. Nevertheless, since the above requirement may not always hold, this straightforward algorithm does not give an FPRAS.

Karp and Luby [KL83] observed that the dependence of N on w can be avoided if, instead of using $\mathcal{A} = \text{Sol}(\varphi)$ and $\mathcal{U} = \{0, 1\}^n$, we use suitably defined alternate sets $\widehat{\mathcal{A}}$ and $\widehat{\mathcal{U}}$, while ensuring that $|\text{Sol}(\varphi)| = |\widehat{\mathcal{A}}|$ and $\frac{1}{\rho} = |\widehat{\mathcal{U}}|/|\widehat{\mathcal{A}}|$ is bounded polynomially in m and n . This is a powerful technique and researchers have used this idea and its variants over the years to achieve improvement in runtime performance and quality of approximations of Monte Carlo DNF counters. We mention below three variants of Monte Carlo approximate DNF counting algorithms.

Karp and Luby's counter Karp and Luby [KL83] developed the first FPRAS for DNF counting, which we refer to as KL Counter. They defined a new universe

$\widehat{\mathcal{U}} = \{(\sigma, \varphi^i) \mid \sigma \models \varphi^i\}$, and the corresponding solution space $\widehat{\mathcal{A}} = \{(\sigma, \varphi^i) \mid \sigma \models \varphi^i \text{ and } \forall j < i, \sigma \not\models \varphi^j\}$ for a fixed ordering of the cubes. They showed that $|\text{Sol}(\varphi)| = |\widehat{\mathcal{A}}|$ and the ratio $\frac{|\widehat{\mathcal{U}}|}{|\widehat{\mathcal{A}}|} \leq m$. Furthermore, executing Step 3 of Algorithm 3 takes $\mathcal{O}(n)$ time, while executing Step 4 takes $\mathcal{O}(mn)$ time. Consequently, the time complexity of KL Counter is in $\mathcal{O}(\frac{m^2 n^2}{\varepsilon^2} \cdot \log(\frac{1}{\delta}))$. Karp and Luby showed that by designing a slightly different unbiased estimator, this complexity can indeed be reduced to $\mathcal{O}(\frac{m^2 n}{\varepsilon^2} \log(\frac{1}{\delta}))$.

Karp, Luby and Madras' counter Karp, Luby and Madras [KLM89] proposed an improvement of KL Counter by employing a non 0-1 estimator. Towards this end, the concept of *coverage* of an assignment σ in \mathcal{U} was introduced as $\text{cover}(\sigma) = \{j \mid \sigma \models \varphi^j\}$. The first key insight of Karp et al. was that $|\widehat{\mathcal{A}}| = \sum_{(\sigma, \varphi^i) \in \widehat{\mathcal{U}}} \frac{1}{|\text{cover}(\sigma)|}$. Their second key insight was to define an estimator for $1/|\text{cover}(\sigma)|$ using the geometric distribution. They were then able to show that the time complexity of the resulting Monte-Carlo algorithm, which we call KLM Counter, is in $\mathcal{O}(\frac{mn}{\varepsilon^2} \cdot \log(\frac{1}{\delta}))$ – an improvement over KL Counter.

Vazirani's counter A variant of KLM Counter, called Vazirani Counter, was described in Vazirani [Vaz13], where $|\text{cover}(\sigma)|$ is computed exactly by iterating over all cubes, avoiding the use of the geometric distribution in [KLM89]. The advantage of Vazirani Counter is that it is able to utilize ideas for optimal Monte Carlo estimation proposed in [DKLR00]. Consequently, Vazirani Counter requires fewer samples than KL Counter to achieve the same error bounds. The time for generating a sample, however, can be considerably more since the check for $\sigma \models \varphi^j$ has to be performed for all cubes.

26.4.2. DNF Counting via Hashing-based Approach

While Monte Carlo techniques have been the predominant paradigm for approximate DNF counting, it turns out that the universal hashing based approach also yields an FPRAS for DNF counting. Recall that Algorithm HASHCOUNTER, described in Section 26.2.1 is a randomized approximation scheme that makes polynomially many invocations to SATURATINGCOUNTER. It has been observed in [CMV16, MSV17] that SATURATINGCOUNTER can be designed to run in time polynomial in the size of the input formula and the threshold, if the input formula is in DNF. To see why this is true, observe that satisfiability checking of $\varphi \wedge h_i^{-1}(0^i)$ is self-reducible if $h_i^{-1}(0^i)$ is a conjunction of XOR constraints and φ is a DNF formula. Furthermore, the satisfiability of $\varphi \wedge h_i^{-1}(0^i)$ can be checked in polynomial time by iterating over all the cubes of the input formula, substituting the forced assignments induced by each cube into the XOR constraints separately, and using Gauss-Jordan Elimination to check satisfiability of the simplified XOR constraints. Interestingly, we can even avoid appealing to the self-reducibility of satisfiability checking of $\varphi \wedge h_i^{-1}(0^i)$ by simply enumerating solutions of the simplified XOR constraints. Note that at no step, one would have to enumerate more than *thresh* solutions. Concretely, this leads to an FPRAS, called DNFApproxMC in [MSV17], for DNF formulas with complexity $\mathcal{O}((mn^3 + mn^2/\varepsilon^2) \log n \log(1/\delta))$.

The existence of universal hashing-based FPRAS for DNF counting leads to an obvious question: *can these algorithms match the asymptotic complexity of Monte Carlo based FPRAS for DNF counting?* Towards this end, Meel et al. [MSV17] sought to avoid the need for Gauss-Jordan elimination originating due to the usage of \mathcal{H}_{xor} or $\mathcal{H}_{\mathcal{T}}$. Specifically, they proposed a new class of hash functions \mathcal{H}_{Rex} such that every hash function $h \in \mathcal{H}_{\text{Rex}}$ that maps $\{0, 1\}^n \mapsto \{0, 1\}^m$ can be represented as $\mathbf{A}x + \mathbf{b}$, where \mathbf{A} is a 0-1 random matrix of dimension $m \times n$ in row-echelon form and \mathbf{b} is a random 0-1 matrix of dimension $m \times 1$. In particular, we can represent \mathbf{A} as $\mathbf{A} = [\mathbf{I} : \mathbf{D}]$, where \mathbf{I} is the identity matrix of size $m \times m$ and \mathbf{D} is a random 0-1 matrix of size $m \times (n - m)$. In follow-up work, Meel et al. [MSV19] sought to avoid the complexity bottleneck presented by the requirement of having to enumerate up to `thresh` solutions in every call to `SATURATINGCOUNTER`. Towards this end, they proposed the search for the right number (m) of cells in reverse order starting from $m = n$ instead of from $m = 0$. This helps in ensuring that only $\mathcal{O}(\text{thresh})$ solutions are enumerated during the entire iteration of the loop in lines 7–15 of Algorithm `HASHCOUNTER` (see Algorithm 1).

In a series of papers [MSV17, MSV19], Meel et al. proposed several other improvements, eventually resulting in a universal hashing-based DNF counting algorithm, called `SymbolicDNFApproxMC`, that runs in time $\tilde{O}(\frac{mn}{\varepsilon^2} \log(1/\delta))$. Note that this matches the asymptotic complexity bound for Monte Carlo techniques discussed above. One would expect that the asymptotic complexity of different FPRAS correlates positively with their empirical run-time behavior when comparing the performance of different FPRAS on a large set of benchmarks. In a rather surprising twist, however, the landscape of empirically observed run-time behavior turns out to be far more nuanced than that captured by worst-case analysis, as shown in [MSV19]. In particular, there is no single algorithm that outperforms all others for all classes of formulas and input parameters. Interestingly, Meel et al. observed that the algorithm, `DNFApproxMC`, with one of the worst-time complexities ended up solving the largest number of benchmarks.

It is worth noting that there has been a long gap of 34 years between Karp and Luby's Monte Carlo-based FPRAS for DNF counting, and the discovery of hashing-based FPRAS for DNF counting. This is despite the fact that hashing techniques for CNF counting have been known at least since Stockmeyer's seminal work in 1983. Interestingly, Stockmeyer's technique for transforming an algorithm with a constant-factor approximation to one with an $(1 + \varepsilon)$ -factor approximation does not work for DNF counting. This is because conjoining a set of DNF formulas (a key component of Stockmeyer's technique) does not always yield a formula representable in DNF without an exponential blow-up. In contrast, in hashing-based approximate counting (ref. Algorithm `HASHCOUNTER`), a $(1 + \varepsilon)$ -factor approximation is easily achieved by choosing `thresh` to be in $\mathcal{O}(1/\varepsilon^2)$.

26.5. Weighted Counting

A natural extension of model counting is to augment the formula φ with a weight function ρ that assigns a non-negative weight to every assignment of values to variables. The problem of weighted model counting, also known as discrete in-

tegration, seeks to compute the weight of φ , defined as the sum of the weight of all its solutions. Note that if ρ assigns weight 1 to each assignment, then the corresponding problem is simply model counting.

Formally, let $\rho : \{0, 1\}^n \rightarrow \mathbb{Q}_+ \cap [0, 1]$ be a *weight function* mapping each truth assignment to a non-negative rational number in $[0, 1]$ such that (i) $\forall \sigma \in \{0, 1\}^n$, $\rho(\sigma)$ is computable in polynomial time, and (ii) $\forall \sigma \in \{0, 1\}^n$, $\rho(\sigma)$ is written in binary representation with p bits. We extend the weight function to sets of truth assignments and Boolean formulas in the obvious way. If Y is a subset of $\{0, 1\}^n$, the weight of Y is defined as the cumulative weight of the truth assignments in Y : $\rho(Y) = \sum_{\sigma \in Y} \rho(\sigma)$. By definition, the weight of the empty set is 0. The weight of a formula φ is defined as the cumulative weight of its solutions, i.e., $\rho(\varphi) = \sum_{\sigma \models \varphi} \rho(\sigma)$. Given ρ and φ , the problem of weighted model counting seeks to compute $\rho(\varphi)$. The polynomial-time computability of $\rho(\sigma)$ implies that the problem of weighted counting is #P-complete.

The recent success of approximate (unweighted) model counters has inspired several attempts for approximate weighted model counting as well [EGSS13a, EGSS13b, CFM⁺14, EGSS14, AJ15, AD16, dCM19]. Despite impressive advances made by these attempts, the inherent hardness of weighted model counting has repeatedly manifested itself as a scalability challenge. This highlights the need and opportunity for both algorithmic and systems-oriented research in this area.

In order to present a unified view of different algorithms proposed for weighted model counting, we follow the treatment of Ermon et al [EGSS13b] and de Colnet and Meel [dCM19], and introduce two additional notions: the *tail function* and *effective weight function*. The tail function τ maps the space of weights (i.e. $\mathbb{Q}_+ \cap [0, 1]$) to \mathbb{N} . Informally, $\tau(u)$ counts the number of models of φ with weight at least as large as u . Formally, $\tau(u) = |\{\sigma \in \{0, 1\}^n \mid \sigma \models \varphi \text{ and } \rho(\sigma) \geq u\}|$. The effective weight function w , which is the dual of the tail function, can be intuitively thought of as mapping non-zero natural numbers to weights. Informally, $w(t)$ gives the largest weight of a model σ of φ that has at least t models of larger or equal weight. For technical reasons, we formally define w as a mapping from the positive reals (instead of positive natural numbers) to the space of weights as follows: $w(t) = \max_{\sigma} \{w(\sigma) \mid \sigma \models \varphi \text{ and } \tau(\rho(\sigma)) \geq t\}$. Note that both the effective weight function and the tail function are non-increasing functions of their argument. It can be shown [EGSS13b, dCM19] that the weighted model count $\rho(\varphi)$ is exactly the area under the $\tau(u)$ (plotted against u) curve, as well as that under the $w(t)$ (plotted against t) curve. In other words, $\rho(\varphi) = \int \tau(u) du = \int w(t) dt$. Both of these are integrals of non-increasing functions defined over \mathbb{R}_+ and are of finite support.

Recent work on approximate weighted model counting can now be broadly described as following a two-step strategy. In the first step, the task of weighted counting is reduced to an integration problem for a suitable real-valued non-increasing function. In the second step, methods based on upper and lower rectangle approximations or Monte Carlo integrators are used to approximate the integral of a real function. We describe below two different approaches that follow this strategy, effectively reducing weighted counting to optimization and unweighted counting respectively.

26.5.1. From weighted counting to optimization

Ermon et al. [EGSS13a, EGSS13b] used the rectangle approximation of $\int \tau(u)du$ to reduce the problem of weighted counting to one of optimization. The first step in this process is the partitioning of the weight (i.e., u) axis into $\mathcal{O}(n)$ intervals, where $n = |\text{Sup}(\varphi)|$. Specifically, the axis is split at the weights $q_0, q_1 \dots, q_n$, where q_i is the maximal weight such that $\tau(q_i) \geq 2^i$; we call these q_i 's *quantile weights*. Observe that the quantile weights are all well-defined, and form a non-increasing sequence. Furthermore, for each positive quantile weight q_i , there exists some truth assignment such that $q_i = \rho(\sigma)$.

The partitioning of the u axis in $\int \tau(u)du$ at the quantile weights gives $\rho(\varphi) = q_n 2^n + \sum_{i=1}^n \int_{q_i}^{q_{i-1}} \tau(u)du$ where $\int_{q_i}^{q_{i-1}}$ represents the integral on $(q_i, q_{i-1}]$. Note that if u is in $(q_i, q_{i-1}]$, then we have $2^{i-1} \leq \tau(u) \leq 2^i$. Therefore, $2^{i-1} (q_{i-1} - q_i) \leq \int_{q_i}^{q_{i-1}} \tau(u)du \leq 2^i (q_{i-1} - q_i)$. Summing all bounds together and rearranging the terms, we obtain $q_0 + \sum_{i=0}^{n-1} q_{i+1} 2^i \leq \rho(\varphi) \leq q_0 + \sum_{i=0}^{n-1} q_i 2^i$. Thus, if W_1 denotes $q_0 + \sum_{i=0}^{n-1} q_{i+1} 2^i$, we have $W_1 \leq \rho(\varphi) \leq 2W_1$.

Given q_0, \dots, q_n , the estimate W_1 can be computed in polynomial time. Recall that for all i , the weight q_i is defined as $\max \{ \rho(\sigma) \mid \sigma \models \varphi \text{ and } \tau(\rho(\sigma)) \geq 2^i \}$. Therefore the task of approximating the weighted model count $\rho(\varphi)$ has effectively been transformed to $n + 1$ optimization problems. While computing q_i exactly is intractable, we can obtain a good approximation of q_i via the usage of 2-universal hash functions. Similar to hashing-based counting, the core technical idea is to employ 2-universal hash functions to partition the space of solutions, viz. $\{0, 1\}^n$, into approximately equal-sized 2^i cells. We then choose an arbitrary cell and use an optimization query (viz. MPE query in [EGSS13b] or MaxSAT query in [dCM19]) to find a solution of φ with maximum weight in the chosen cell. Let us call this σ_{max} . The 2-universality property of the hash family ensures that $\rho(\sigma_{max}) \in [q_{i-2}, q_{i+2}]$ with probability greater than $\frac{1}{2}$. Therefore, we can choose $\rho(\sigma_{max})$ as an estimate \hat{q}_i of q_i . As in the case of counting, the above process of randomly partitioning the space of solutions and finding an estimate of q_i can be repeated an appropriate number (linear in $\log \frac{1}{\delta}$ and $\log n$) of times, and the median of the estimates used with high enough confidence. The estimate \widehat{W}_1 of W_1 computed using the median estimates of q_i s can be shown to be a 16-factor approximation with confidence at least $1 - \delta$ [EGSS13b, dCM19]. To obtain a $(1 + \varepsilon)$ -factor approximation, Stockmeyer's technique, as outlined in Section 26.2.3.1, can be applied.

26.5.2. From weighted counting to counting

We now discuss techniques that effectively use rectangular approximations of $\int w(t)dt$ to estimate a weighted model count. The work of Chakraborty et al. [CFM⁺14] and de Colnet and Meel [dCM19] belong to this category. As in the previous subsection, the first step is to partition the tail (i.e., t) axis. Towards this end, we choose a non-decreasing sequence $\tau_0 \leq \tau_1 \leq \dots \leq \tau_p$ such that $\tau_i = \tau(1/2^i)$ and p is the number of bits in binary representation of $\rho(\sigma)$ as mentioned in Section 26.5. We call these τ_i 's *splitting tails*.

The partitioning of the integral $\int w(t)dt$ at the splitting tails gives $\rho(\varphi) = \tau_0 + \sum_{i=0}^{p-1} \int_{\tau_i}^{\tau_{i+1}} w(t)dt$, where $\int_{\tau_i}^{\tau_{i+1}}$ represents the integral on $(\tau_i, \tau_{i+1}]$. If t is in $(\tau_i, \tau_{i+1}]$, then $2^{-i-1} \leq w(t) \leq 2^{-i}$. So we bound the tails in each interval $(\tau_i, \tau_{i+1}]$ within a factor of 2 as $2^{-i-1}(\tau_{i+1} - \tau_i) \leq \int_{\tau_i}^{\tau_{i+1}} w(t)dt \leq 2^{-i}(\tau_{i+1} - \tau_i)$. Note that the bound holds even when $(\tau_i, \tau_{i+1}]$ is empty (i.e. $\tau_i = \tau_{i+1}$). Summing all bounds together and rearranging the terms, we obtain $\tau_p 2^{-p} + \sum_{i=0}^{p-1} \tau_i 2^{-(i+1)} \leq \rho(\varphi) \leq \tau_p 2^{-p} + \sum_{i=0}^{p-1} \tau_{i+1} 2^{-(i+1)}$. Let W_2 denote $\tau_p 2^{-p} + \sum_{i=0}^{p-1} \tau_i 2^{-(i+1)}$. Then, we have $W_2 \leq \rho(\varphi) \leq 2W_2$. As in the previous subsection, if we are given $\{\tau_0, \dots, \tau_p\}$, the estimate W_2 can be computed in polynomial time. Furthermore, recall that the tail τ_i is defined to be $|\{\sigma \mid \sigma \models \varphi \text{ and } \rho(\sigma) \geq 2^{-i}\}|$. Therefore, the problem of estimating $\rho(\varphi)$ is now transformed to $p + 1$ counting sub-problems. Note that each sub-problem requires us to count not just models of φ , but models with a specified minimum weight. In special cases, where the weight of an assignment is the product or sum of weights of corresponding literals, the set of models of φ with weight at least 2^{-i} can be represented using a Pseudo-Boolean constraint, as has been done in [CFM⁺14]. Estimating the model count of such a constraint can be accomplished using the same hashing-based technique discussed earlier, by partitioning the solution space using random XOR constraints. Let $\hat{\tau}_i$ denote the estimate of τ_i obtained in this manner. Chakraborty et al. [CFM⁺14] and de Colnet and Meel [dCM19] have shown that the estimate \widehat{W}_2 of W_2 obtained by using $\hat{\tau}_i$ instead of τ_i is a constant-factor approximation with confidence at least $1 - \delta$ [dCM19]. As before, Stockmeyer’s technique can be used to obtain a $(1 + \varepsilon)$ -factor approximation of $\rho(\varphi)$.

In a separate line of work, Chakraborty et al. [CFMV15] and Dudek, Fried, and Meel [DFM20] investigated reduction from weighted model counting to unweighted model counting for a specific class of weight functions called *literal weight functions*. In this case, weights are assigned to individual literals, and the weight of an assignment is simply the product of weights of its literals. For a variable x_i in the support of φ and a weight function $\rho(\cdot)$, let $\rho(x_i^1)$ and $\rho(x_i^0)$ denote the weights of the positive and negative literals respectively, of x_i . Without loss of generality, we assume that $\rho(x_i^1) + \rho(x_i^0) = 1$, and that each $\rho(x_i^1) = \frac{k_i}{\ell_i}$, where $k_i, \ell_i \in \mathbb{N}$ and $k_i \leq \ell_i$. The key idea in [CFMV15, DFM20] is to transform a given formula φ into another formula $\widehat{\varphi}$ such that $\rho(\varphi) = C_\rho \cdot |\text{Sol}(\widehat{\varphi})|$, where C_ρ depends only on ℓ_i ’s, and *not* on φ . Once the above transformation is achieved, the problem of estimating $\rho(\varphi)$ reduces to that of counting models of $\widehat{\varphi}$. To achieve the transformation, Chakraborty et al used a gadget called *chain formulas* [CFMV15]. The chain formula ψ_{k_i, ℓ_i} represents a Boolean formula with k_i solutions over $\lceil \log_2 \ell_i \rceil$ fresh variables. The transformed formula $\widehat{\varphi}$ is then obtained as $\varphi \wedge \bigwedge_{i=1}^n ((x_i \rightarrow \psi_{k_i, \ell_i}) \wedge (\neg x_i \rightarrow \psi_{\ell_i - k_i, \ell_i}))$. An approximate counting technique, viz. Algorithm HASHCOUNTER, can now be invoked to obtain a $(1 + \varepsilon)$ -approximation of $|\text{Sol}(\widehat{\varphi})|$ with confidence $1 - \delta$. Since C_ρ is polynomially computable from the ℓ_i ’s, this gives us a PAC estimate of $\rho(\varphi) = C_\rho \cdot |\text{Sol}(\widehat{\varphi})|$.

26.5.3. Scalability challenges

Both the algorithmic approaches to weighted counting outlined above provide rigorous theoretical guarantees on the quality of computed estimates. They have

been implemented and demonstrated to scale to problems involving several thousand propositional variables. Nevertheless, there are factors that have impeded scaling them up to even larger problem sizes. Some of these factors include:

1. The lack of highly efficient MaxSAT or MPE solvers capable of handling constraints with XOR clauses.
2. The lack of solvers capable of handling formulas expressed as a conjunction of CNF, Pseudo-Boolean constraints, and XOR constraints [PJM19].
3. The reduction proposed by Chakraborty et al. [CFMV15] suffers from a linear increase in the number of variables. This results in large XOR constraints when using hash-based approximate counting techniques. Large XOR constraints are known to present scalability hurdles to state-of-the-art satisfiability solvers.

Given these bottlenecks, the spectacular progress witnessed in recent years in MaxSAT, MPE and pseudo-Boolean constraint solving present significant opportunities for pushing the frontiers of the weighted model counting. In addition, the use of short XOR clauses without weakening theoretical guarantees and the integration of Gaussian elimination in MaxSAT, MPE and pseudo-Boolean solvers (akin to what has been done for CNF solvers) present promising research directions to pursue.

26.6. Conclusion

Model counting is a fundamental computational problem with a wide variety of applications. It is not surprising therefore that it has attracted the attention of theoreticians and practitioners for the past four decades. This has resulted in a rich bouquet of algorithms with differing characteristics in terms of guarantees on the quality of estimates and runtime performance. Among them, universal hashing-based approximate counting has emerged as the dominant paradigm over the past decade. The success of hashing-based counters can be attributed to two primary reasons: (i) properties of universal hash functions that allow these algorithms to provide (ε, δ) guarantees, and (ii) the spectacular advances in modern satisfiability solving techniques that allow counting algorithms to scale to constraints involving hundreds of thousands of variables.

While we focused on counting all models of a formula φ in this chapter, the hashing-based techniques described in the preceding sections also generalize to the related problem of projected model counting, wherein given $X \subseteq \text{Sup}(\varphi)$, the task is to compute $|\text{Sol}(\exists X \varphi)|$ [SM19]. Given that usage of the existential quantifier (\exists) can give exponentially succinct formulas vis-a-vis equivalent formulas written without quantifiers, the problem of projected model counting has been employed in several applications ranging from network (un)reliability to neural network verification [DOMPV17, BSS⁺19, NSM⁺19].

The promise exhibited by state-of-the-art approximate model counters has led to the study of several extensions and generalizations: maximum model counting [FRS17], weighted model integration [BPVdB15, KMS⁺18], stochastic satisfiability [LWJ17], and counting over string and bit-vector constraints [LSSD14, ABB15, CMMV16, CDM17]. Each of these generalizations have necessitated the

development of new techniques and have attracted newer applications. Furthermore, approximate model counting and almost-uniform sampling are known to be inter-reducible [JVV86]. The close relationship has led to a long line of fruitful work extending several hashing-based techniques discussed in this chapter to the context of uniform sampling [CMV13a, CMV14, CFM⁺14, CFM⁺15, MVC⁺16, Meel17].

The promise and progress in the development of approximate model counters has opened several new directions of research, that is evident from the recent flurry of activity in the research community studying approximate counting and its applications. The availability of implementations that both scale to large inputs and also provide strong approximation guarantees has ignited interest among practitioners in seeking new applications for counting. Therefore, we look optimistically to the future where the groundwork laid by hashing-based techniques will lead to the development of new paradigms and practical applications of approximate model counting.

References

- [AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge Univ. Press, 2009.
- [ABB15] Abdalbaki Aydin, Lucas Bang, and Tefvik Bultan. Automata-based model counting for string constraints. In *Proc. of CAV*, pages 255–272. Springer, 2015.
- [AD16] Megasthenis Asteris and Alexandros G Dimakis. LDPC codes for discrete integration. Technical report, Technical report, UT Austin, 2016.
- [AHT18] Dimitris Achlioptas, Zayd Hammoudeh, and Panos Theodoropoulos. Fast and flexible probabilistic model counting. In *Proc. of SAT*, pages 148–164. Springer, 2018.
- [AJ15] Dimitris Achlioptas and Pei Jiang. Stochastic integration via error-correcting codes. In *UAI*, pages 22–31, 2015.
- [AT17] Dimitris Achlioptas and Panos Theodoropoulos. Probabilistic model counting with short XORs. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 3–19. Springer, 2017.
- [Bal88] José Luis Balcázar. *Self-reducibility structures and solutions of NP problems*. Departament de Llenguatges i Sistemes Informàtics [de la] Universitat, 1988.
- [BEH⁺18] Fabrizio Biondi, Michael A. Enescu, Annelie Heuser, Axel Legay, Kuldeep S. Meel, and Jean Quilbeuf. Scalable approximation of quantitative information flow in programs. In *Proc. of VMCAI*, 2018.
- [Bet56] Evert W Beth. On Padoa’s method in the theory of definition. *Journal of Symbolic Logic*, 21(2):194–195, 1956.
- [BGP00] Mihir Bellare, Oded Goldreich, and Erez Petrank. Uniform generation of NP-witnesses using an NP-oracle. *Information and Computation*, 163(2):510–526, 2000.

- [BM00] Peter Baumgartner and Fabio Massacci. The taming of the (X)OR. In *Proc. of CL*, pages 508–522. Springer, 2000.
- [BPVdB15] Vaishak Belle, Andrea Passerini, and Guy Van den Broeck. Probabilistic inference in hybrid domains by weighted model integration. In *Proc. of IJCAI*, pages 2770–2776, 2015.
- [BSS⁺19] Teodora Baluta, Shiqi Shen, Shweta Shine, Kuldeep S. Meel, and Prateek Saxena. Quantitative verification of neural networks and its security applications. In *Proc. of CCS*, 2019.
- [CDM17] Dmitry Chistikov, Rayna Dimitrova, and Rupak Majumdar. Approximate counting in SMT and value estimation for probabilistic programs. *Acta Informatica*, 54(8):729–764, 2017.
- [CFM⁺14] Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. Distribution-aware sampling and weighted model counting for SAT. In *Proc. of AAAI*, pages 1722–1730, 2014.
- [CFM⁺15] Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. On parallel scalable uniform sat witness generation. In *Proc. of TACAS*, pages 304–319, 2015.
- [CFMV15] Supratik Chakraborty, Dror Fried, Kuldeep S Meel, and Moshe Y Vardi. From weighted to unweighted model counting. In *Proc. of AAAI*, pages 689–695, 2015.
- [Che09] Jingchao Chen. Building a hybrid SAT solver via conflict-driven, look-ahead and XOR reasoning techniques. In *Proc. of SAT*, pages 298–311. Springer, 2009.
- [CMMV16] Supratik Chakraborty, Kuldeep S. Meel, Rakesh Mistry, and Moshe Y. Vardi. Approximate probabilistic inference via word-level counting. In *Proc. of AAAI*, 2016.
- [CMV13a] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. A scalable and nearly uniform generator of SAT witnesses. In *Proc. of CAV*, pages 608–623, 2013.
- [CMV13b] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. A scalable approximate model counter. In *Proc. of CP*, pages 200–216, 2013.
- [CMV14] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Balancing scalability and uniformity in SAT witness generator. In *Proc. of DAC*, pages 1–6, 2014.
- [CMV16] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT calls. In *Proc. of IJCAI*, 2016.
- [CMV19] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. On the hardness of probabilistic inference relaxations. In *Proc. of AAAI*, 2019.
- [CW77] J Lawrence Carter and Mark N Wegman. Universal classes of hash functions. In *Proc. of STOC*, pages 106–112. ACM, 1977.
- [dCM19] Alexis de Colnet and Kuldeep S. Meel. Dual hashing-based algorithms for discrete integration. In *Proc. of CP*, 10 2019.

- [DFM20] Jeffrey Dudek, Dror Fried, and Kuldeep S. Meel. Taming discrete integration via the boon of dimensionality. In *Proc. of NeurIPS*, 2020.
- [DH07] Carmen Domshlak and Jörg Hoffmann. Probabilistic planning via heuristic forward search and weighted model counting. *Journal of Artificial Intelligence Research*, 30(1):565–620, 2007.
- [DKLR00] Paul Dagum, Richard M. Karp, Michael Luby, and Sheldon Ross. An optimal algorithm for Monte Carlo estimation. *SIAM Journal on Computing*, 29(5):1484–1496, 2000.
- [DOMPV17] Leonardo Duenas-Osorio, Kuldeep S Meel, Roger Paredes, and Moshe Y Vardi. Counting-based reliability estimation for power-transmission grids. In *Proc. of AAAI*, pages 4488–4494, 2017.
- [EGSS13a] Stefano Ermon, Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Optimization with parity constraints: From binary codes to discrete integration. In *Proc. of UAI*, 2013.
- [EGSS13b] Stefano Ermon, Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Taming the curse of dimensionality: Discrete integration by hashing and optimization. In *Proc. of ICML*, pages 334–342, 2013.
- [EGSS14] Stefano Ermon, Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Low-density parity constraints for hashing-based discrete integration. In *Proc. of ICML*, pages 271–279, 2014.
- [FHO13] Robert Fink, Jiewen Huang, and Dan Olteanu. Anytime approximation in probabilistic databases. *The VLDB Journal*, 22(6):823–848, Dec 2013.
- [FRS17] Daniel J Fremont, Markus N Rabe, and Sanjit A Seshia. Maximum model counting. In *Proc. of AAAI*, 2017.
- [GHSS07a] Carla P Gomes, Jörg Hoffmann, Ashish Sabharwal, and Bart Selman. From sampling to model counting. In *Proc. of IJCAI*, pages 2293–2299, 2007.
- [GHSS07b] Carla P. Gomes, Jörg Hoffmann, Ashish Sabharwal, and Bart Selman. Short XORs for model counting: From theory to practice. In *Proc. of SAT*, pages 100–106, 2007.
- [Gra06] Robert M. Gray. *Toeplitz And Circulant Matrices: A Review (Foundations and Trends(R) in Communications and Information Theory)*. Now Publishers Inc., Hanover, MA, USA, 2006.
- [GSS06] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model counting: A new strategy for obtaining good bounds. In *Proc. of AAAI*, volume 21, pages 54–61, 2006.
- [HDVZVM04] Marijn Heule, Mark Dufour, Joris Van Zwieten, and Hans Van Maaren. March.eq: Implementing additional reasoning into an efficient look-ahead sat solver. In *Proc. of SAT*, pages 345–359. Springer, 2004.
- [HJ12] Cheng-Shen Han and Jie-Hong Roland Jiang. When Boolean satisfiability meets Gaussian elimination in a simplex way. In *Proc. of CAV*, pages 410–426. Springer, 2012.
- [HJ19] Mark Huber and Bo Jones. Faster estimates of the mean of bounded random variables. *Mathematics and Computers in Sim-*

- ulation, 161:93–101, 2019.
- [Hub17] Mark Huber. A Bernoulli mean estimate with known relative error distribution. *Random Structures and Algorithms*, 50(2):173–182, 2017.
- [IMMV15] Alexander Ivrii, Sharad Malik, Kuldeep S. Meel, and Moshe Y. Vardi. On computing minimal independent support and its applications to sampling and counting. *Constraints*, pages 1–18, 2015.
- [JVV86] Mark R. Jerrum, Leslie G. Valiant, and Vijay V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theoretical Computer Science*, 43(2-3):169–188, 1986.
- [KL83] Richard M. Karp and Michael Luby. Monte-carlo algorithms for enumeration and reliability problems. *Proc. of FOCS*, 1983.
- [KL85] Richard M. Karp and Michael Luby. Monte-carlo algorithms for the planar multiterminal network reliability problem. *Journal of Complexity*, 1(1):45–64, 1985.
- [KLM89] Richard M. Karp, Michael Luby, and Neal Madras. Monte-carlo approximation algorithms for enumeration problems. *Journal of Algorithms*, 10(3):429 – 448, 1989.
- [KMS⁺18] Samuel Kolb, Martin Mladenov, Scott Sanner, Vaishak Belle, and Kristian Kersting. Efficient symbolic integration for probabilistic inference. In *Proc. of IJCAI*, pages 5031–5037, 2018.
- [LJN10] Tero Laitinen, Tommi Junttila, and Ilkka Niemelä. Extending clause learning DPLL with parity reasoning. In *Proc. of ECAI*, volume 2010, pages 21–26, 2010.
- [LJN11] Tero Laitinen, Tommi Junttila, and Ilkka Niemela. Equivalence class based parity reasoning with DPLL (XOR). In *Prof. of ICTAI*, pages 649–658. IEEE, 2011.
- [LJN12] Tero Laitinen, Tommi Junttila, and Ilkka Niemelä. Conflict-driven XOR-clause learning. In *Proc. of SAT*, pages 383–396. Springer, 2012.
- [LLM16] Jean-Marie Lagniez, Emmanuel Lonca, and Pierre Marquis. Improving model counting by leveraging definability. In *Proc. of IJCAI*, pages 751–757, 2016.
- [LM08] Jérôme Lang and Pierre Marquis. On propositional definability. *Artificial Intelligence*, 172(8-9):991–1017, 2008.
- [LS08] Mark H. Liffiton and Kareem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1):1–33, 2008.
- [LSS17] Jingcheng Liu, Alistair Sinclair, and Piyush Srivastava. The Ising partition function: Zeros and deterministic approximation. *CoRR*, abs/1704.06493, 2017.
- [LSSD14] Loi Luu, Shweta Shinde, Prateek Saxena, and Brian Demsky. A model counter for constraints over unbounded strings. *ACM SIGPLAN Notices*, 49(6):565–576, 2014.
- [LWJ17] Nian-Ze Lee, Yen-Shi Wang, and Jie-Hong R Jiang. Solving stochastic Boolean satisfiability under Random-Exist quantification. In *Proc. of IJCAI*, pages 688–694, 2017.

- [MA20] Kuldeep S Meel and S Akshay. Sparse hashing for scalable approximate model counting: Theory and practice. In *Proc. of LICS*, pages 728–741, 2020.
- [Mee17] Kuldeep S. Meel. *Constrained Counting and Sampling: Bridging the Gap between Theory and Practice*. PhD thesis, Rice University, 2017.
- [MSV17] Kuldeep S. Meel, Aditya A. Shrotri, and Moshe Y. Vardi. On hashing-based approaches to approximate DNF-counting. In *Proc. of FSTTCS*, 12 2017.
- [MSV19] Kuldeep S. Meel, Aditya A. Shrotri, and Moshe Y. Vardi. Not all FPRASs are equal: Demystifying FPRASs for DNF-counting. *Constraints*, 24(3-4):211–233, 2019.
- [Mur68] Katta G. Murty. An algorithm for ranking all the assignments in order of increasing cost. *Operations Research*, 16(3):682–687, 1968.
- [MVC⁺16] Kuldeep S Meel, Moshe Vardi, Supratik Chakraborty, Daniel J Fremont, Sanjit A Seshia, Dror Fried, Alexander Ivrii, and Sharad Malik. Constrained sampling and counting: Universal hashing meets SAT solving. In *Proc. of Beyond NP Workshop*, 2016.
- [Nad10] Alexander Nadel. Boosting minimal unsatisfiable core extraction. In *Proc. of FMCAD*, pages 221–229, 2010.
- [NSM⁺19] Nina Narodytska, Aditya A. Shrotri, Kuldeep S. Meel, Alexey Ignatiev, and Joao Marques-Silva. Assessing heuristic machine learning explanations with model counting. In *Proc. of SAT*, pages 267–278, 2019.
- [PJM19] Yash Pote, Saurabh Joshi, and Kuldeep S. Meel. Phase transition behavior of cardinality and XOR constraints. In *Proc. of IJCAI*, 8 2019.
- [R⁺70] Alfréd Rényi. *Probability Theory*. North Holland, Amsterdam, 1970.
- [Rot96] Dan Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1):273–302, 1996.
- [SGM20] Mate Soos, Stephan Gocht, and Kuldeep S Meel. Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling. In *Proc. of CAV*, pages 463–484. Springer, 2020.
- [SM19] Mate Soos and Kuldeep S Meel. BIRD: Engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In *Proc. of AAAI*, 2019.
- [SNC09] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *Proc. of SAT*, pages 244–257, 2009.
- [Soo10] Mate Soos. Enhanced Gaussian elimination in DPLL-based SAT solvers. In *Proc. of Pragmatics of SAT Workshop*, pages 2–14, 2010.
- [Sto83] Larry Stockmeyer. The complexity of approximate counting. In *Proc. of STOC*, pages 118–126, 1983.
- [SVP⁺16] Somdeb Sarkhel, Deepak Venugopal, Tuan Anh Pham, Parag

- Singla, and Vibhav Gogate. Scalable training of Markov logic networks using approximate counting. In *Proc. of AAAI*, pages 1067–1073, 2016.
- [Tod89] Seinosuke Toda. On the computational power of PP and (+)P. In *Proc. of FOCS*, pages 514–519. IEEE, 1989.
- [Val79] Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.
- [Vaz13] Vijay V. Vazirani. *Approximation algorithms*. Springer Science & Business Media, 2013.
- [WS05] Wei Wei and Bart Selman. A new approach to model counting. In *Proc. of SAT*, pages 2293–2299. Springer, 2005.
- [XCD12] Yexiang Xue, Arthur Choi, and Adnan Darwiche. Basing decisions on sentences in decision diagrams. In *Proc. of AAAI*, 2012.
- [ZCSE16] Shengjia Zhao, Sorathan Chaturapuek, Ashish Sabharwal, and Stefano Ermon. Closing the gap between short and long XORs for model counting. In *Proc. of AAAI*, 2016.
- [ZQRZ18] Ziqiao Zhou, Zhiyun Qian, Michael K Reiter, and Yinqian Zhang. Static evaluation of noninterference using approximate model counting. In *Proc. of IEEE Symposium on Security and Privacy*, pages 514–528. IEEE, 2018.

This page intentionally left blank

Chapter 27

Non-Clausal SAT and ATPG

Rolf Drechsler, Tommi Junttila and Ilkka Niemelä

27.1. Introduction

When studying the propositional satisfiability problem (SAT), that is, the problem of deciding whether a propositional formula is satisfiable, it is typically assumed that the formula is given in the conjunctive normal form (CNF). Also most software tools for deciding satisfiability of a formula (SAT solvers) assume that their input is in CNF. An important reason for this is that it is simpler to develop efficient data structures and algorithms for CNF than for arbitrary formulas. On the other hand, using CNF makes efficient modeling of an application cumbersome. Therefore one often employs a more general formula representation in modeling and then transforms the formula into CNF for SAT solvers. Transforming a propositional formula to an equivalent formula in CNF can increase the size of the formula exponentially in the worst case. However, general propositional formulas can be transformed in polynomial time into CNF while preserving the satisfiability of the instance [PG86, BdlT92, NRW98, JS05, Vel04] (also see Chapter 2 in this handbook). Such translations introduce auxiliary variables which can have an exponential effect on the performance of a SAT solver in the worst-case. Moreover, by translating to CNF one often loses information about the structure of the original problem.

In this chapter we survey methods for solving propositional satisfiability problems when the input formula is not given in a normal form and techniques in a closely related area of automatic test pattern generation (ATPG). Substantial amount of research has been done on developing theorem proving methods working with arbitrary (non-clausal) formulas. A major part of this work is based on tableaux type of calculi [DGHE99, Häh01] but this research has mostly considered the first-order case or non-classical logics. For a comprehensive summary of tableau type calculi for classical propositional logic, see [D'A99]. There has been some work on developing efficient propositional provers based on tableau techniques such as the HARP prover [OS88]. However, in the classical propositional case tableau based approaches have not been as successful as DPLL based techniques (see Chapter 3 in this handbook for more details on the DPLL algorithm). Typical tableau calculi for classical propositional logic are not able to

simulate polynomially even truth tables [DM94] but there has been some work on extending tableau techniques to make them computationally more efficient, e.g., by introducing an explicit cut rule like in the KE system [DM94] or by adding new constraint propagation rules [Mas98]. An interesting generalization of tableaux is provided by dissolution [MR93]. Another major approach to developing a computationally attractive method for deciding satisfiability of a general formula is based on Stålmarck's proof procedure for propositional logic [SS00] (see Chapter 3 in this handbook for more details on Stålmarck's algorithm). In fact, a commercial SAT solver system, *Prover*, is based on this method [Bor97]. Also local search methods for satisfiability have been extended to work on the non-clausal case, e.g., in [Seb94, KMS97, PTS07]. A very successful approach to manipulate general propositional formulas is based on Binary Decision Diagrams (BDDs) [Bry86, Bry92]. However, BDDs are not targeted towards satisfiability checking but provide an attractive normal form that is especially suitable for determining equivalence of formulas.

A substantial number of advanced satisfiability checking techniques that work on non-clausal representation have been developed in the area of digital circuit design. These approaches work typically with *Boolean circuits*. This work is closely related to the area of automatic test pattern generation (ATPG) for digital circuits where similar sophisticated techniques have been developed.

Circuit representation has properties that can be exploited in satisfiability checking in a number of ways: Circuits preserve important structural information and allow sharing of common sub-expressions enabling very concise problem representation. These properties can be used in preprocessing and for efficient Boolean propagation. Moreover, circuit representation enables to detect observability don't cares which are very useful in pruning the search space in satisfiability checking. In addition, circuit structure can be used to guide search heuristics.

In this chapter we survey satisfiability checking methods that work directly on Boolean circuits and techniques for ATPG for digital circuits. In these areas there is a substantial body of literature which is covered in the corresponding sections. The rest of the chapter is structured as follows. We start by introducing Boolean circuits in Section 27.2. Then in Section 27.3 we present methods for Boolean circuit satisfiability checking focusing on successful DPLL type clausal SAT checking techniques generalized and extended to work directly with Boolean circuits. Section 27.4 reviews classical ATPG algorithms, formulation of ATPG as a SAT problem, and advanced techniques for SAT-based ATPG.

27.2. Basic Definitions

Assume the set of Booleans $\mathbb{B} = \{\mathbf{F}, \mathbf{T}\}$ and define $\neg\mathbf{F} = \mathbf{T}$ and $\neg\mathbf{T} = \mathbf{F}$.

A *Boolean circuit* \mathcal{C} is a pair $(\mathcal{G}, \mathcal{E})$, where

- \mathcal{G} is a non-empty, finite set of *gates*, and
- \mathcal{E} is a set of equations such that
 - each equation is of form $g := f(g_1, \dots, g_n)$, where $g, g_1, \dots, g_n \in \mathcal{G}$ and $f \in [\mathbb{B}^n \rightarrow \mathbb{B}]$ is a Boolean function,
 - each gate $g \in \mathcal{G}$ appears at most once as the left hand side in the equations in \mathcal{E} , and

- the equations are not recursive, that is, the directed *dependency graph* $\text{graph}(\mathcal{C}) = (\mathcal{G}, \{(g', g) \mid g := f(\dots, g', \dots) \in \mathcal{E}\})$ is acyclic.

Notice that we use here a definition of Boolean circuits similar to that in [Pap95] where signals of a circuit \mathcal{C} are not represented explicitly but are given implicitly as edges of the graph $\text{graph}(\mathcal{C})$. Figure 27.1 clarifies the correspondence between the graph representation $\text{graph}(\mathcal{C})$ of a Boolean circuit \mathcal{C} and the representation with signals given explicitly.

A gate $g \in \mathcal{G}$ is a *primary input* gate if it does not appear as the left hand side in any equation in \mathcal{E} ; the set of all primary input gates in \mathcal{C} is denoted by $\text{inputs}(\mathcal{C})$. A gate g' is a *child* of a gate g if the edge (g', g) appears in $\text{graph}(\mathcal{C})$, i.e. if g' appears in the equation of g . In this case, g is called the *parent* of g' .¹ The number of children (parents, respectively) of a gate is called the *fan-in* (*fan-out*, respectively) of the gate. The concepts of *descendant* and *ancestor* gates are defined in the intuitive way via the transitive closures of the child and parent relations, respectively. A gate is a *primary output* gate if it has no parents. The following Boolean function families are commonly used in the gates:

- **false, true** : \mathbb{B} are the constant functions with $\text{false}() = \mathbf{F}$ and $\text{true}() = \mathbf{T}$.
- **not** : $\mathbb{B} \rightarrow \mathbb{B}$ with $\text{not}(\mathbf{F}) = \mathbf{T}$ and $\text{not}(\mathbf{T}) = \mathbf{F}$.
- **and** : $\mathbb{B}^n \rightarrow \mathbb{B}$ with $\text{and}(v_1, \dots, v_n) = \mathbf{T}$ iff all v_1, \dots, v_n are \mathbf{T} .
- **or** : $\mathbb{B}^n \rightarrow \mathbb{B}$ with $\text{or}(v_1, \dots, v_n) = \mathbf{T}$ iff at least one of v_1, \dots, v_n is \mathbf{T} .
- **ite** : $\mathbb{B}^3 \rightarrow \mathbb{B}$ is the “if-then-else” function such that $\text{ite}(v_1, v_2, v_3) = \mathbf{T}$ iff (i) $v_1 = \mathbf{T}$ and $v_2 = \mathbf{T}$, or (ii) $v_1 = \mathbf{F}$ and $v_3 = \mathbf{T}$.
- **odd** : $\mathbb{B}^n \rightarrow \mathbb{B}$ is the n -ary parity function such that $\text{odd}(v_1, \dots, v_n) = \mathbf{T}$ iff an odd number of v_1, \dots, v_n are \mathbf{T} ; the case $n = 2$ is usually called the exclusive-or (**xor**) function.
- **equiv** : $\mathbb{B}^n \rightarrow \mathbb{B}$ is the n -ary equivalence function; $\text{equiv}(v_1, \dots, v_n) = \mathbf{T}$ iff either (i) all v_1, \dots, v_n are \mathbf{T} , or (ii) all v_1, \dots, v_n are \mathbf{F} .

Example 27.2.1. Figure 27.1 shows two standard graphical representations for a Boolean circuit describing a full-adder. Formally, the circuit is defined as $\mathcal{C} = (\mathcal{G}, \mathcal{E})$ with the gate set $\mathcal{G} = \{a_0, b_0, c_0, t_1, t_2, t_3, o_0, c_1\}$ and equations

$$\mathcal{E} = \{c_1 := \text{or}(t_1, t_2), t_1 := \text{and}(t_3, c_0), o_0 := \text{odd}(t_3, c_0), \\ t_2 := \text{and}(a_0, b_0), t_3 := \text{odd}(a_0, b_0)\}.$$

For instance, the gate c_1 is an **odd-gate**, a_0 is a primary input gate, and c_1 is a primary output gate. The gate t_3 is a child of the gate t_1 and a descendant of the gate c_1 .

A *truth assignment* for \mathcal{C} is a possibly partial function $\tau : \mathcal{G} \rightarrow \mathbb{B}$. A gate $g \in \mathcal{G}$ is *assigned* in τ if $\tau(g)$ is defined, otherwise it is *unassigned* in τ ; τ is *total* if all gates are assigned in it. If τ is a truth assignment for \mathcal{C} , $g \in \mathcal{G}$, and $v \in \mathbb{B}$, then $\tau[g \rightarrow v]$ is the truth assignment that is equal to τ except that $\tau[g \rightarrow v](g) = v$. A truth assignment τ' is an *extension* of a truth assignment τ if $\tau' \supseteq \tau$. A total truth assignment τ for \mathcal{C} is *consistent* if $\tau(g) = f(\tau(g_1), \dots, \tau(g_n))$ holds for each

¹The terms input and output of a gate are sometimes used instead of child and parent, respectively; we use the latter ones to avoid confusion with primary inputs and outputs.

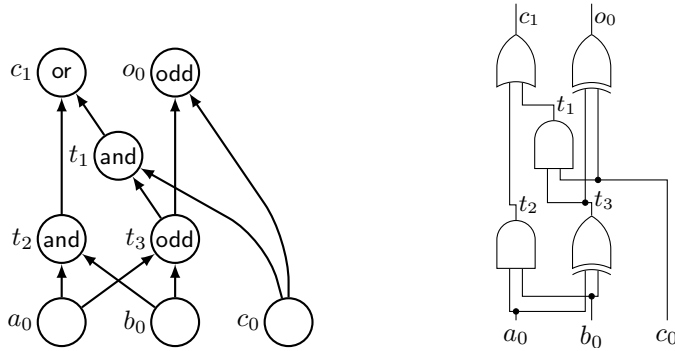


Figure 27.1. Two graphical representations for a full-adder circuit

equation $g := f(g_1, \dots, g_n) \in \mathcal{E}$. Obviously, there are $2^{|\text{inputs}(\mathcal{C})|}$ consistent total truth assignments for \mathcal{C} .

Example 27.2.2. Consider again the circuit \mathcal{C} in Figure 27.1. In the truth assignment $\tau = \{c_1 \mapsto \mathbf{T}, o_0 \mapsto \mathbf{F}\}$, the gate c_1 is assigned and t_1 is unassigned. The truth assignment

$$\tau' = \{c_1 \mapsto \mathbf{T}, t_1 \mapsto \mathbf{F}, o_0 \mapsto \mathbf{F}, t_2 \mapsto \mathbf{T}, t_3 \mapsto \mathbf{F}, a_0 \mapsto \mathbf{T}, b_0 \mapsto \mathbf{T}, c_0 \mapsto \mathbf{F}\}$$

is a total and consistent extension of τ . On the other hand, no total extension of the truth assignment $\{c_1 \mapsto \mathbf{T}, a_0 \mapsto \mathbf{T}, b_0 \mapsto \mathbf{T}, t_3 \mapsto \mathbf{T}\}$ is consistent as the definition of the gate t_3 will not be respected.

The value of a gate g assigned in a truth assignment τ is *justified* in τ if (i) g is a primary input gate, or (ii) $g := f(g_1, \dots, g_n) \in \mathcal{E}$ and $\tau(g) = f(\tau'(g_1), \dots, \tau'(g_n))$ holds for each total truth assignment $\tau' \supseteq \tau$. That is, the value of a justified gate stays consistent with respect to the definition of the gate, no matter how the truth assignment is extended. By definition it holds that if τ is a total and consistent truth assignment for \mathcal{C} , then the value of each gate is justified in τ .

Example 27.2.3. In the truth assignment $\tau = \{c_1 \mapsto \mathbf{T}, t_1 \mapsto \mathbf{T}, t_3 \mapsto \mathbf{T}\}$ for the circuit in Figure 27.1, the gate c_1 is justified. On the other hand, the gates $t_1, t_2,$ and t_3 are not justified in τ .

A *constrained Boolean circuit* is a pair (\mathcal{C}, τ) , where \mathcal{C} is a Boolean circuit and τ is a truth assignment for \mathcal{C} . A total truth assignment τ' for \mathcal{C} *satisfies* (\mathcal{C}, τ) if it is a consistent total extension of τ . A constrained Boolean circuit (\mathcal{C}, τ) is *satisfiable* if there is a total truth assignment for \mathcal{C} that satisfies (\mathcal{C}, τ) , otherwise (\mathcal{C}, τ) is *unsatisfiable*. Obviously, it is an **NP**-complete problem to decide whether there is a satisfying truth assignment for a given constrained circuit.²

Example 27.2.4. Consider again the circuit \mathcal{C} in Figure 27.1. The pair (\mathcal{C}, τ) , where $\tau = \{c_1 \mapsto \mathbf{T}\}$, is a constrained Boolean circuit with the requirement that

²Provided that all the families of Boolean functions attached to gates are computable in polynomial time; this is assumed in the rest of the chapter.

the carry-out gate c_1 should evaluate to true. It is satisfiable as the total assignment $\tau' = \{c_1 \mapsto \mathbf{T}, t_1 \mapsto \mathbf{F}, o_0 \mapsto \mathbf{F}, t_2 \mapsto \mathbf{T}, t_3 \mapsto \mathbf{F}, a_0 \mapsto \mathbf{T}, b_0 \mapsto \mathbf{T}, c_0 \mapsto \mathbf{F}\}$ satisfies it.

27.3. Satisfiability Checking for Boolean Circuits

In this section we survey satisfiability checking techniques which work on the Boolean circuit representation. We start by a brief discussion on preprocessing and then in Section 27.3.2 outline how the highly successful DPLL method for checking satisfiability of a formula in CNF (see Chapter 3 in this handbook) can be generalized to work directly with circuits. In Section 27.3.3 we discuss how clause learning techniques can be integrated into circuit level DPLL methods. We compare Boolean Constraint Propagation (BCP) techniques on the circuit level to those used in clausal level in Section 27.3.4 and elaborate efficient implementation techniques of circuit level BCP in Section 27.3.5. Exploiting observability don't cares is discussed in Section 27.3.6. Section 27.3.7 considers search heuristics exploiting the circuit structure and Section 27.3.8 briefly reviews incomplete local search techniques for Boolean circuit satisfiability.

27.3.1. Preprocessing

When solving a circuit satisfiability problem it is often useful to simplify the circuit with preprocessing techniques preserving satisfiability. Here we briefly discuss some techniques taking advantage of the circuit representation.

A key difference to the flat CNF representation is that circuits allow *sharing of common sub-expressions*. That is, if a circuit has two similar equations, $g := f(g_1, \dots, g_n)$ and $g' := f(g_1, \dots, g_n)$, then one of them, say, $g' := f(g_1, \dots, g_n)$ can be removed from the circuit and all the occurrences of g' can be substituted with g .

Sharing of common sub-expressions can be enhanced and a more compact representation obtained by translating a circuit to some efficient *reduced representation* such as Boolean Expression Diagrams (BEDs) [AH02, WAH03], Reduced Boolean Circuits (RBCs) [ABE00], and And-Inverter Graphs (AIGs) [KGP01, KPKG02]. In addition, logic synthesis and minimization techniques can be applied to further reduce the circuit size, see e.g. [EMS07].

When considering a constrained circuit, where certain gates have been assigned truth values, further simplifications are enabled, for instance, in two ways:

- *Propagation and rewriting based on assigned truth values*
Naturally, the values of the assigned gates can propagate values to other gates. For instance, if an **and**-gate is assigned to **T**, then all its children can be assigned to **T**. In addition, if a gate has an assigned child, the connection to the child can be removed and the gate definition changed accordingly. For example, if for a gate $g := \text{ite}(g_1, g_2, g_3)$ the truth value of the child g_2 is set to **T**, then the equation can be simplified to $g := \text{or}(g_1, g_3)$.
- *Cone of influence reduction*
If a gate is not constrained and no other equation refers to it (i.e. it is an unconstrained primary output gate), it can be removed.

27.3.2. Basic Tableau Method for Circuit Satisfiability Checking

In the circuit satisfiability problem we are given a constrained Boolean circuit (\mathcal{C}, τ) and the task is to determine whether there is a total truth assignment satisfying (\mathcal{C}, τ) . We start by presenting a circuit satisfiability checking method which generalizes the DPLL method for clausal satisfiability to circuits. We formulate the circuit level DPLL method as a tableau system, which we called **BC**, following the approach in [JN00]. First we discuss a basic sound and complete version of the tableau system and then extend it with additional rules enhancing its efficiency.

The basic system consists of the rules shown in Figure 27.2. Note that the versions of the rules obtained by permuting the children of gates with symmetric Boolean functions are not shown in Figures 27.2 and 27.3; for instance,

$$\frac{g := \text{odd}(g_1, g_2, g_3) \quad \mathbf{T}g_1, \mathbf{T}g_3 \quad \mathbf{F}g_2}{\mathbf{F}g}$$

obtained by instantiating the rightmost rule in Figure 27.2(g) for $k = 3$ is also a rule. Given a constrained Boolean circuit (\mathcal{C}, τ) , a **BC-tableau** for (\mathcal{C}, τ) is a binary tree where

- the root node of the tree consists of (i) the equations \mathcal{E} in $\mathcal{C} = (\mathcal{G}, \mathcal{E})$ and (ii) for each gate g assigned in τ , the entry $\mathbf{T}g$ if $\tau(g) = \mathbf{T}$ or $\mathbf{F}g$ if $\tau(g) = \mathbf{F}$;
- the other nodes in the tree are entries of the form $\mathbf{T}g$ or $\mathbf{F}g$, where $g \in \mathcal{G}$, generated by extending the tableau using the tableau rules in the standard way [D'A99]: given a tableau rule and a branch in the tableau such that the prerequisites of the rule hold in the branch, the tableau can be extended by adding new nodes to the end of the branch as specified by the rule.

If the tableau is extended with the explicit cut rule (Figure 27.2 (a)), then entries $\mathbf{T}g$ and $\mathbf{F}g$ are added as the left and right child in the end of the branch. For the other, *non-branching rules* the consequents of the rule are added to the end of the branch (as a linear sub-tree in case of multiple consequents).

A branch in the tableau is *contradictory* if it contains both $\mathbf{F}g$ and $\mathbf{T}g$ entries for a gate $g \in \mathcal{G}$. Otherwise, the branch is *open*. A branch is *complete* if it is contradictory, or if there is a $\mathbf{F}g$ or a $\mathbf{T}g$ entry for each $g \in \mathcal{G}$ in the branch and the branch is closed under the “up” rules (b–g) in Figure 27.2. A tableau is *finished* if all the branches of the tableau are complete. A tableau is *closed* if all of its branches are contradictory. A closed **BC-tableau** for a constrained circuit is called a **BC-refutation** for the circuit.

The basic tableau system **BC** can be shown to be a *refutationally sound and complete* proof system in the sense that there is a **BC-refutation** (closed tableau) for a constrained circuit iff the circuit is unsatisfiable. Also for finding satisfying truth assignments the tableau system is

- *sound* in the sense that if there is an open complete branch in a tableau, then the branch provides a total truth assignment satisfying the constrained circuit and

$$\frac{g \in \mathcal{G}}{\mathbf{T}g \mid \mathbf{F}g} \qquad \frac{g := \text{true}()}{\mathbf{T}g} \qquad \frac{g := \text{false}()}{\mathbf{F}g} \qquad \frac{g := \text{not}(g_1)}{\mathbf{F}g_1} \qquad \frac{g := \text{not}(g_1)}{\mathbf{T}g_1}$$

(a) The explicit cut rule. (b) Constant rules. (c) “Up” rules for not-gates.

$$\frac{g := \text{or}(g_1, \dots, g_k)}{\mathbf{F}g_1, \dots, \mathbf{F}g_k} \qquad \frac{g := \text{and}(g_1, \dots, g_k)}{\mathbf{T}g_1, \dots, \mathbf{T}g_k} \qquad \frac{g := \text{or}(g_1, \dots, g_k)}{\mathbf{T}g_i, i \in \{1, \dots, k\}} \qquad \frac{g := \text{and}(g_1, \dots, g_k)}{\mathbf{F}g_i, i \in \{1, \dots, k\}}$$

(d) “Up” rules for or- and and-gates.

$$\frac{g := \text{ite}(g_1, g_2, g_3)}{\mathbf{T}g_1, \mathbf{T}g_2} \qquad \frac{g := \text{ite}(g_1, g_2, g_3)}{\mathbf{F}g_1, \mathbf{T}g_3} \qquad \frac{g := \text{ite}(g_1, g_2, g_3)}{\mathbf{T}g_1, \mathbf{F}g_2} \qquad \frac{g := \text{ite}(g_1, g_2, g_3)}{\mathbf{F}g_1, \mathbf{F}g_3}$$

(e) “Up” rules for ite-gates.

$$\frac{g := \text{equiv}(g_1, \dots, g_k)}{\mathbf{T}g_1, \dots, \mathbf{T}g_k} \qquad \frac{g := \text{equiv}(g_1, \dots, g_k)}{\mathbf{F}g_1, \dots, \mathbf{F}g_k} \qquad \frac{g := \text{equiv}(g_1, \dots, g_k)}{\mathbf{T}g_i, i \in \{1, \dots, k\}} \qquad \frac{g := \text{equiv}(g_1, \dots, g_k)}{\mathbf{F}g_j, j \in \{1, \dots, k\}}$$

(f) “Up” rules for equiv-gates.

$$\frac{g := \text{odd}(g_1, \dots, g_k)}{\mathbf{T}g_1, \dots, \mathbf{T}g_j, j \text{ is odd}} \qquad \frac{g := \text{odd}(g_1, \dots, g_k)}{\mathbf{T}g_1, \dots, \mathbf{T}g_j, j \text{ is even}} \qquad \frac{g := \text{odd}(g_1, \dots, g_k)}{\mathbf{F}g_{j+1}, \dots, \mathbf{F}g_k} \qquad \frac{g := \text{odd}(g_1, \dots, g_k)}{\mathbf{F}g_{j+1}, \dots, \mathbf{F}g_k}$$

(g) “Up” rules for odd-gates.

Figure 27.2. Basic rules.

- *complete* in the sense that if the circuit is satisfiable, then any finished tableau contains an open branch that provides a total truth assignment satisfying the constrained circuit.

In order to enhance the efficiency of the tableau system additional rules given in Figure 27.3 can be added without compromising the soundness and completeness of the system. For a gate $g \in \mathcal{G}$, we say that an entry $\mathbf{T}g$ ($\mathbf{F}g$) can be *deduced* in a branch if the entry $\mathbf{T}g$ ($\mathbf{F}g$) can be generated by applying non-branching rules, i.e., other rules than the cut rule (Figure 27.2 (a)).

Example 27.3.1. Consider again the circuit \mathcal{C} in Figure 27.1 and constraints $\tau = \{c_1 \mapsto \mathbf{T}, b_0 \mapsto \mathbf{F}\}$. A BC-tableau for the constrained circuit (\mathcal{C}, τ) is shown in Figure 27.4. For instance, the entry 8. $\mathbf{F}t_1$ has been generated by applying the explicit cut rule (Figure 27.2 (a)) and the entry 9. $\mathbf{T}t_2$ has been deduced from entries 1. $c_1 := \text{or}(t_1, t_2)$, 6. $\mathbf{T}c_1$, and 8. $\mathbf{F}t_1$ using the “Indirect down” rule for or-gates (Figure 27.3 (b)).

The tableau is finished and the left branch is contradictory (marked with \times) as it contains entries $\mathbf{F}b_0$ (7) and $\mathbf{T}b_0$ (11). However, the right branch is open

$$\begin{array}{c}
g := \text{not}(g_1) \quad g := \text{not}(g_1) \quad g := \text{or}(g_1, \dots, g_k) \quad g := \text{and}(g_1, \dots, g_k) \\
\frac{\mathbf{T}g}{\mathbf{F}g_1} \quad \frac{\mathbf{F}g}{\mathbf{T}g_1} \quad \frac{\mathbf{F}g}{\mathbf{F}g_1, \dots, \mathbf{F}g_k} \quad \frac{\mathbf{T}g}{\mathbf{T}g_1, \dots, \mathbf{T}g_k}
\end{array}$$

(a) “Direct down” rules for not-, or-, and and-gates.

$$\begin{array}{c}
g := \text{or}(g_1, \dots, g_k) \quad g := \text{and}(g_1, \dots, g_k) \\
\frac{\mathbf{T}g}{\mathbf{F}g_1, \dots, \mathbf{F}g_{k-1}} \quad \frac{\mathbf{F}g}{\mathbf{T}g_1, \dots, \mathbf{T}g_{k-1}} \\
\frac{\mathbf{F}g_1, \dots, \mathbf{F}g_{k-1}}{\mathbf{T}g_k} \quad \frac{\mathbf{T}g_1, \dots, \mathbf{T}g_{k-1}}{\mathbf{F}g_k}
\end{array}$$

$$\begin{array}{c}
g := \text{ite}(g_1, g_2, g_3) \quad g := \text{ite}(g_1, g_2, g_3) \\
\frac{\mathbf{T}g}{\mathbf{T}g_1} \quad \frac{\mathbf{F}g}{\mathbf{T}g_1} \\
\frac{\mathbf{T}g_1}{\mathbf{T}g_2} \quad \frac{\mathbf{F}g_1}{\mathbf{T}g_3}
\end{array}$$

$$\begin{array}{c}
g := \text{ite}(g_1, g_2, g_3) \quad g := \text{ite}(g_1, g_2, g_3) \\
\frac{\mathbf{T}g}{\mathbf{F}g_1, \mathbf{T}g_3} \quad \frac{\mathbf{T}g}{\mathbf{F}g_2, \mathbf{T}g_3} \\
\frac{\mathbf{F}g_2, \mathbf{T}g_3}{\mathbf{T}g} \quad \frac{\mathbf{F}g_2, \mathbf{F}g_3}{\mathbf{F}g}
\end{array}$$

(b) “Indirect down” rules for or and and. (c) Redundant “up” rules for ite-gates.

$$\begin{array}{c}
g := \text{ite}(g_1, g_2, g_3) \quad g := \text{ite}(g_1, g_2, g_3) \quad g := \text{ite}(g_1, g_2, g_3) \quad g := \text{ite}(g_1, g_2, g_3) \\
\frac{\mathbf{T}g}{\mathbf{T}g_1} \quad \frac{\mathbf{T}g}{\mathbf{F}g_1} \quad \frac{\mathbf{T}g}{\mathbf{F}g_2} \quad \frac{\mathbf{T}g}{\mathbf{F}g_3} \\
\frac{\mathbf{T}g_1}{\mathbf{T}g_2} \quad \frac{\mathbf{F}g_1}{\mathbf{T}g_3} \quad \frac{\mathbf{F}g_1, \mathbf{T}g_3}{\mathbf{F}g_2, \mathbf{T}g_3} \quad \frac{\mathbf{T}g_1, \mathbf{T}g_2}{\mathbf{T}g_1, \mathbf{T}g_2}
\end{array}$$

$$\begin{array}{c}
g := \text{ite}(g_1, g_2, g_3) \quad g := \text{ite}(g_1, g_2, g_3) \quad g := \text{ite}(g_1, g_2, g_3) \quad g := \text{ite}(g_1, g_2, g_3) \\
\frac{\mathbf{F}g}{\mathbf{T}g_1} \quad \frac{\mathbf{F}g}{\mathbf{F}g_1} \quad \frac{\mathbf{F}g}{\mathbf{T}g_2} \quad \frac{\mathbf{F}g}{\mathbf{T}g_3} \\
\frac{\mathbf{F}g_1}{\mathbf{F}g_2} \quad \frac{\mathbf{F}g_1}{\mathbf{F}g_3} \quad \frac{\mathbf{F}g_1, \mathbf{F}g_3}{\mathbf{F}g_1, \mathbf{F}g_3} \quad \frac{\mathbf{T}g_2}{\mathbf{T}g_1, \mathbf{F}g_2}
\end{array}$$

(d) “Indirect down” rules for ite-gates.

$$\begin{array}{c}
g := \text{equiv}(g_1, \dots, g_k) \quad g := \text{equiv}(g_1, \dots, g_k) \\
\frac{\mathbf{T}g}{\mathbf{T}g_i, i \in \{1, \dots, k\}} \quad \frac{\mathbf{T}g}{\mathbf{F}g_i, i \in \{1, \dots, k\}} \\
\frac{\mathbf{T}g_i, i \in \{1, \dots, k\}}{\mathbf{T}g_1, \dots, \mathbf{T}g_k} \quad \frac{\mathbf{F}g_i, i \in \{1, \dots, k\}}{\mathbf{F}g_1, \dots, \mathbf{F}g_k}
\end{array}$$

$$\begin{array}{c}
g := \text{equiv}(g_1, \dots, g_k) \quad g := \text{equiv}(g_1, \dots, g_k) \\
\frac{\mathbf{F}g}{\mathbf{T}g_1, \dots, \mathbf{T}g_{k-1}} \quad \frac{\mathbf{F}g}{\mathbf{F}g_1, \dots, \mathbf{F}g_{k-1}} \\
\frac{\mathbf{T}g_1, \dots, \mathbf{T}g_{k-1}}{\mathbf{F}g_k} \quad \frac{\mathbf{F}g_1, \dots, \mathbf{F}g_{k-1}}{\mathbf{T}g_k}
\end{array}$$

(e) “Indirect down” rules for equiv-gates.

$$\begin{array}{c}
g := \text{odd}(g_1, \dots, g_k) \quad g := \text{odd}(g_1, \dots, g_k) \\
\frac{\mathbf{T}g}{\mathbf{T}g_1, \dots, \mathbf{T}g_j, j \text{ is odd}} \quad \frac{\mathbf{F}g}{\mathbf{T}g_1, \dots, \mathbf{T}g_j, j \text{ is odd}} \\
\frac{\mathbf{F}g_{j+1}, \dots, \mathbf{F}g_{k-1}}{\mathbf{F}g_k} \quad \frac{\mathbf{F}g_{j+1}, \dots, \mathbf{F}g_{k-1}}{\mathbf{T}g_k}
\end{array}$$

$$\begin{array}{c}
g := \text{odd}(g_1, \dots, g_k) \quad g := \text{odd}(g_1, \dots, g_k) \\
\frac{\mathbf{T}g}{\mathbf{T}g_1, \dots, \mathbf{T}g_j, j \text{ is even}} \quad \frac{\mathbf{F}g}{\mathbf{T}g_1, \dots, \mathbf{T}g_j, j \text{ is even}} \\
\frac{\mathbf{F}g_{j+1}, \dots, \mathbf{F}g_{k-1}}{\mathbf{T}g_k} \quad \frac{\mathbf{F}g_{j+1}, \dots, \mathbf{F}g_{k-1}}{\mathbf{F}g_k}
\end{array}$$

(f) “Indirect down” rules for odd-gates.

Figure 27.3. Additional rules.

and it gives a truth assignment

$$\{c_1 \mapsto \mathbf{T}, b_0 \mapsto \mathbf{F}, t_1 \mapsto \mathbf{T}, t_3 \mapsto \mathbf{T}, c_0 \mapsto \mathbf{T}, o_0 \mapsto \mathbf{F}, a_0 \mapsto \mathbf{T}, t_2 \mapsto \mathbf{F}\}.$$

that satisfies (\mathcal{C}, τ) .

1. $c_1 := \text{or}(t_1, t_2)$	
2. $t_1 := \text{and}(t_3, c_0)$	
3. $o_0 := \text{odd}(t_3, c_0)$	
4. $t_2 := \text{and}(a_0, b_0)$	
5. $t_3 := \text{odd}(a_0, b_0)$	
6. $\mathbf{T}c_1$	
7. $\mathbf{F}b_0$	
8. $\mathbf{F}t_1$ (Cut)	13. $\mathbf{T}t_1$ (Cut)
9. $\mathbf{T}t_2$ (1, 6, 8)	14. $\mathbf{T}t_3$ (2, 13)
10. $\mathbf{T}a_0$ (4, 9)	15. $\mathbf{T}c_0$ (2, 13)
11. $\mathbf{T}b_0$ (4, 9)	16. $\mathbf{F}o_0$ (3, 14, 15)
12. \times (7, 11)	17. $\mathbf{T}a_0$ (5, 7, 14)
	18. $\mathbf{F}t_2$ (4, 7, 17)

Figure 27.4. A BC-tableau for (\mathcal{C}, τ)

The tableau system **BC** can be seen as a generalization of DPLL to Boolean circuits. First, the explicit cut rule (Figure 27.2 (a)) corresponds to the branching step in DPLL. Second, for Boolean constraint propagation (BCP) the non-branching rules (Figure 27.2 (b–g) and Figure 27.3) play the role of unit propagation (unit resolution) used in DPLL. In Section 27.3.4 we discuss the relationship of BCP using non-branching rules on the circuit level and clausal BCP in more detail.

27.3.3. Implication Graphs, Conflict Driven Learning, and Non-Chronological Backtracking

Conflict driven learning and non-chronological backtracking (backjumping), see Chapters 3 and 4 in this handbook, are techniques applied in state-of-the-art CNF-based SAT solvers to reduce the search spaces. They are based on analyzing each conflict encountered during the search to find (i) a new conflict clause that, when learned (that is, added to the CNF formula), will prevent similar conflicts from occurring again, and (ii) information about which parts of the current branch were irrelevant for the conflict and can thus be jumped over when backtracking. The conflict analysis is based on implication graphs that capture how decisions and Boolean constraint propagation were performed in the branch under consideration.

Integrating conflict driven learning and backjumping into circuit level DPLL is quite straightforward if we interpret the non-branching rules above as implications. For instance, the leftmost “up” rule in Figure 27.2 for an odd-gate $g := \text{odd}(g_1, g_2, g_3)$ can be read as “if g_1 is false, g_2 is true, and g_3 is false, then g

is true”, or thought as a clause $(\neg g_1 \wedge g_2 \wedge \neg g_3 \rightarrow g) \equiv (g_1 \vee \neg g_2 \vee g_3 \vee g)$. Because of this interpretation, one can track the use of the non-branching rules during the DPLL search to build an implication graph and derive a conflict clause from it in a way similar to that in Chapter 3 of this handbook. The conflict clause $(g'_1 \vee \dots \vee g'_l \vee \neg g''_{l+1} \vee \dots \vee \neg g''_m)$ can then be added (either explicitly or implicitly, see Section 27.3.5) to the circuit as the or-gate $g' := \text{or}(g'_1, \dots, g'_l, g''_{l+1}, \dots, g''_m)$ that is constrained to **T** and where $g''_j := \text{not}(g'_j)$ for $l + 1 \leq j \leq m$. The addition of such constrained gates derived by the conflict analysis is sound in the sense that all satisfying truth assignments of the constrained circuit are preserved.

27.3.4. Boolean Constraint Propagation: Rules versus Clauses

We now briefly compare circuit and CNF level Boolean constraint propagation. Let us first consider the non-branching rules for an or-gate $g := \text{or}(g_1, \dots, g_k)$ in Figures 27.2 and 27.3 with the clauses

$$(\hat{g} \vee \neg \hat{g}_1), \dots, (\hat{g} \vee \neg \hat{g}_k), (\neg \hat{g} \vee \hat{g}_1 \vee \dots \vee \hat{g}_k)$$

in the corresponding Tseitin CNF translation where the variable \hat{g} corresponds to the gate g , \hat{g}_1 to g_1 and so on (c.f. Chapter 2 in this handbook). Note that the clauses are nothing but the rules interpreted as clauses as in Section 27.3.3. It is obvious that the rules and the clauses have a very close correspondence when considering Boolean constraint propagation: a rule is applicable if and only if the “corresponding” clause induces unit propagation in corresponding partial truth assignments. A similar close correspondence applies to not and and-gates, too, although it should be mentioned that not-gates are not usually translated to CNF gates but “inlined” to the signs of the literals in the CNF formula. The not-gates are also handled implicitly in some circuit based propagation techniques by extending the edges in the circuit with sign information (e.g. in the And-Inverter Graph based approach of [KPKG02]).

The situation is somewhat different for odd-gates with n children as the straightforward CNF translation obtained by simply interpreting the rules as implications has 2^n clauses; for instance, the CNF translation of $g := \text{odd}(g_1, g_2, g_3)$ would be

$$\begin{aligned} &(\hat{g}_1 \vee \hat{g}_2 \vee \hat{g}_3 \vee \neg \hat{g}) \wedge (\hat{g}_1 \vee \hat{g}_2 \vee \neg \hat{g}_3 \vee \hat{g}) \wedge \\ &(\hat{g}_1 \vee \neg \hat{g}_2 \vee \hat{g}_3 \vee \hat{g}) \wedge (\hat{g}_1 \vee \neg \hat{g}_2 \vee \neg \hat{g}_3 \vee \neg \hat{g}) \wedge \\ &(\neg \hat{g}_1 \vee \hat{g}_2 \vee \hat{g}_3 \vee \hat{g}) \wedge (\neg \hat{g}_1 \vee \hat{g}_2 \vee \neg \hat{g}_3 \vee \neg \hat{g}) \wedge \\ &(\neg \hat{g}_1 \vee \neg \hat{g}_2 \vee \hat{g}_3 \vee \neg \hat{g}) \wedge (\neg \hat{g}_1 \vee \neg \hat{g}_2 \vee \neg \hat{g}_3 \vee \hat{g}). \end{aligned}$$

However, it is well-known that by (virtually) rewriting an n -ary odd-gate as $n - 1$ binary odd-gates and then translating each of them to CNF with 4 ternary clauses, one can obtain a CNF translation that has $4(n - 1)$ clauses and $n - 1$ auxiliary variables. This translation has the desirable property that if a non-branching rule on a gate $g := \text{odd}(g_1, \dots, g_n)$ is applicable in a partial truth assignment, then unit propagation will produce the same propagation in the corresponding truth assignment in the translated formula.

Similar translations and analyzes can be obtained for equiv and ite-gates, too. In addition to the gate types considered so far, cardinality gates, i.e. gates of the form $g := \text{card}_l^u(g_1, \dots, g_k)$ that evaluate to true iff at least l but at most u of the

children are true, and their sub-classes such as the “at most one child is true”-gate, are of special interest in some applications. Translating such gates to CNF is bit more complicated, see e.g. [BB03, BB04, Sin05, MSL07] and Chapter 28 in this handbook for some approaches.

As a summary, for the gate types considered above it is possible to obtain CNF translations that enable CNF-level unit propagation to simulate circuit-level Boolean constraint propagation. However, if Boolean constraint propagation is implemented directly on the circuit-level (this is discussed in the next subsection), one can take the available gate type information into account when designing data structures and algorithms for BCP. In [JBH12] it is shown that, under the common CNF translation schemes, the CNF preprocessing techniques such as variable elimination and blocked clause elimination can to a large extent simulate circuit level transformations such as cone of influence reduction. On the other hand, circuit level techniques such as observability don't cares discussed in Section 27.3.6 can be applied relatively efficiently during the search as well.

27.3.5. Implementation Techniques for Boolean Constraint Propagation

As Boolean Constraint Propagation (BCP), i.e. applying the non-branching rules in Figures 27.2 and 27.3, lies in the inner loop of a DPLL-based satisfiability checker for circuits, implementing it efficiently is of great importance. Naturally, if one transforms the circuit in question into an equi-satisfiable CNF formula by using the standard Tseitin-translation (cf. Section 27.3.4 and Chapter 2 in this handbook), one can apply all the BCP techniques developed for CNF-based SAT-solvers (see Chapter 4 in this handbook). In the rest of this section we review circuit-based BCP implementation techniques that are not based on direct CNF translation.

27.3.5.1. The Straightforward Way

The most straightforward way to implement BCP is described below. First, a queue (or stack, or unordered set) of assigned but not yet propagated gates is maintained and the following is repeated until the queue becomes empty. First, get and remove a gate from the queue. Then check whether the value of the gate implies a value to an unassigned child by a “direct down” rule (e.g. a true **and**-gate implies that all its children are true); if it does, assign the child to the value and add it in the queue. Next check whether the value of the gate and the values of the already assigned child gates imply a value to an unassigned child by an “indirect down” rule (e.g. a false **and**-gate with all but one child assigned to true imply that the unassigned child is false); if yes, assign and queue the child as described earlier. Finally, for each parent of the gate, check whether any “up” or “indirect down” rule involving the gate is applicable (e.g. if the gate is assigned to true and it has a false **and**-parent with all but one child assigned to true); if yes, apply the rule (e.g. the unassigned child is assigned to false) and queue the newly assigned gate. Naturally, a conflict may occur in any of these cases, e.g. when an **and**-gate is assigned to true but one of its children is already false.

In such a situation, the propagation is canceled and, if conflict driven learning or non-chronological backtracking is applied, the reason for the conflict is analyzed.

This simple propagation technique works in a local way in the circuit. It requires that the list of parents of a gate is maintained in addition to the type and the children of the gate. One drawback is that if the applicability of a non-branching rule depends on the values of the children of a gate, then the values of the children must be fetched each time the applicability is checked. However, backtracking is efficient as only the values of the assigned gates have to be cleared.

27.3.5.2. A Counter Based Approach

To reduce the number of times the values of the children of a gate have to be fetched during BCP, one can assign two counters to each gate with a symmetric Boolean function (i.e. all other gates except *ite*-gates considered in this chapter); similar techniques have been used in the context of CNF-based SAT and stable models logic programming [DG84, CA93, NS96]. The counters are maintained during the propagation process to tell how many of the children are assigned to false and true, respectively. Each time a gate with symmetric Boolean function is assigned to a value, it is easy to check the applicability of the “down” rules by just considering the values of the counters. For instance, if an n -ary *and*-gate is assigned to false, the number of true child gates is $n - 1$, and the number of false children is 0, then the last unassigned child is searched and assigned to false. However, if the number of true children is less than $n - 1$, then nothing is done. In addition, each time a gate is assigned to a value, the counters of the parent gates (with symmetric Boolean functions) are updated and the applicability of the “up” and “indirect down” rules of the parent gates are checked by using the values of counters. Note that backtracking in this counter-based scheme is more time consuming than in the straightforward way described above because the values of the counters have to be decremented in addition to just clearing the assigned values.

27.3.5.3. A Lookup Table Based Approach

Another kind of BCP technique is described in [KGP01, KPKG02]. It assumes the circuits to be composed of primary input, *not*-, and binary *and*-gates only (also called the *And-Inverter Graph* representation for Boolean circuits). As the fan-in of the gates is fixed to a very small constant, it is feasible to enumerate all the possible local propagation cases in a lookup table. Each time a gate is assigned a value, the lookup table for that gate as well as for each of its parents is consulted for finding out the possible propagations. In the case where no propagation is possible, i.e. when an *and*-gate is assigned to false and none of its two children is currently assigned, the gate is appended to the justification queue; the DPLL-like backtracking search process selects the next gate to be branched on by popping a gate from the justification queue and then branches on one of its children, trying to justify the value of the gate (cf. Section 27.3.7.2). In [KGP01, KPKG02] the learned conflict clauses (recall Section 27.3.3) are decomposed into *not*- and binary *and*-gates and added to the circuit structure. Thus, propagation induced by them is handled by the same lookup table mechanism.

27.3.5.4. Watched Literal Techniques

In [GZA⁺02] the lookup table based technique of [KGP01, KPKG02] for binary gates is combined with the “watched literal” technique applied in state-of-the-art CNF-based SAT solvers ([MMZ⁺01], also see Chapter 4 in this handbook). The idea is that the propagation in the original circuit structure is done with lookup tables while the propagation induced by the learned conflict clauses is implemented with the watched literal technique. This is motivated by the fact that learned clauses can contain hundreds of literals and the watched literal technique is efficient especially for such long clauses. Forgetting of learned conflict clauses, a technique applied in clause learning DPLL-style SAT solvers to reduce memory consumption and propagation overhead, is also easier to implement in this scheme as the learned clauses are stored separately while the circuit structure stays unmodified.

In [TBW04] a variant of the CNF-based watched literal technique is introduced for unbounded fan-in and- and or-gates. First, direct propagation (e.g. a true and-gate implies that all its children are true, and a false gate implies that each of its and-parent is false) does not involve watches but is performed in the straightforward way. For an and-gate, two of its children are watched for true value, the invariant to be maintained being that “neither of the watched children should be assigned to true unless there is no other choice or the other watched child is assigned to false”. Each time a watched child becomes assigned to true, it is checked whether the invariant holds (the other watched child is false) or can be made to hold by watching some other child. If this is not possible, then either all or all but one of the children are true. In the first case, the and-gate is propagated to true. In the second case, it is checked whether the and-gate is false; if it is, then the “indirect down” propagation rule is activated and the yet unassigned child is assigned to false; otherwise it is waited until the and-gate obtains a value. When an and-gate is assigned to false, it is checked whether either of the watched children is true; if yes, then all but one of the children are true and thus the remaining unassigned child is propagated to false.

In [WLLH07], the watched literal technique is generalized from CNF-level to Boolean circuits with arbitrary gate types. Instead of watching only the children of a gate like in [TBW04], the gate itself can also be watched. For and- and or-gates the generalized watched literal scheme follows easily from their Tseitin CNF translations. For instance, the CNF translation of an n -ary and-gate $g := \text{and}(g_1, \dots, g_n)$ is composed of n binary clauses of form $(\neg g \vee g_i)$ and one longer clause $(g \vee \neg g_1 \vee \dots \vee \neg g_n)$. The propagation corresponding to the binary clauses (i.e. the non-branching rules with only one antecedent) is handled simply by attaching each gate with two lists of (gate,value) pairs: those that are directly implied when the gate becomes false or true, respectively (for instance, the pair (g_1, \mathbf{T}) would be in the “true list” of g when $g := \text{and}(g_1, \dots, g_n)$). The propagation corresponding to the clause $(g \vee \neg g_1 \vee \dots \vee \neg g_n)$ is performed by watching two of the (gate,value) pairs in $\{(g, \mathbf{F}), (g_1, \mathbf{T}), \dots, (g_n, \mathbf{T})\}$. The invariant to be maintained is that “neither of the watched pairs should be assigned to true (a pair (g, v) is assigned to true [false] if g is assigned to v [$\neg v$, respectively]) or one of the watched pairs is assigned to false”. Whenever a watched (gate,value) pair becomes assigned to true, it is checked whether the invariant can be made

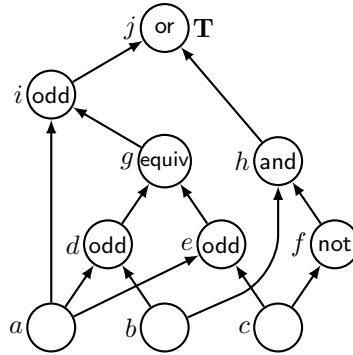


Figure 27.5. A constrained circuit.

to hold by watching some other pair; if it cannot be, then the last remaining unassigned pair is propagated to false. For instance, if all g_1, \dots, g_{n-1} are true and g becomes assigned to false, the only way to make the invariant hold on the set $\{(g, \mathbf{F}), (g_1, \mathbf{T}), \dots, (g_n, \mathbf{T})\}$ is to assign the gate g_n to false. The watched literal technique for other gate types, such as **odd** and **ite**, applies “watching-known pointers”: both values of a gate are watched at the same time. As an example, for an **odd**-gate $g := \text{odd}(g_1, \dots, g_n)$, two of the gates in the set $\{g, g_1, \dots, g_n\}$ are watched. The invariant to be maintained is now “neither of the watched gates should be assigned to any value unless there is no other choice”. Each time a watched gate becomes assigned, one tries to maintain the invariant by watching some other gate; if this is not possible, then propagation is triggered; in our example, when all g, g_1, \dots, g_{n-1} are assigned, then g_n is propagated to the value determined by the equation $g := \text{odd}(g_1, \dots, g_n)$. The paper gives rules to determine how many of the gates should be watched for a given gate type; for instance, **odd**-gates need two watched gates, an **ite** gate $g := \text{ite}(g_i, g_t, g_e)$ requires three of the gates in $\{g, g_i, g_t, g_e\}$ to be watched, and n -ary **equiv**-gates must have n gates watched.

Like with the CNF-based watched literal techniques, backtracking is efficient when using circuit-level watched literal techniques as only the assigned values have to be reset but the watches do not have to be touched at all.

27.3.6. Exploiting Observability Don’t Cares

Observability don’t cares are one of the concepts easily visible in the circuit level but lost in a flat CNF representation. In a nutshell, an observability don’t care is a gate whose value cannot influence the constraints of the circuit under the current partial truth assignment. As an example, consider the constrained Boolean circuit in Figure 27.5. If the first branching in the search assigns the gate i to \mathbf{T} , then the value of the constrained primary output gate j becomes justified in the resulting partial truth assignment. This implies that the value of the gate h cannot influence the value of j (or any other constrained ancestor gate) anymore and thus cannot cause a conflict by upward constraint propagation. Or in other words, the constrained gate j cannot observe the value of the gate h , and

h is called an observability don't care. Observability don't cares can be exploited during the DPLL-search by restricting the branching on the gates that are not don't cares, i.e. that are still relevant for the search. For instance, in the example above one does not need to branch on the gates h and f in the sub-tree originating from the first branching. If at some point of the search all the non-don't care gates are assigned without having a conflict by Boolean constraint propagation, the partial truth assignment can be extended to a satisfying truth assignment by evaluating the don't care gates (if there is a don't care primary input gate, it can be assigned to an arbitrary value in the extension).

As noted in [SVDL04], don't cares can also “mask” earlier, possibly bad, decisions made during the search. For instance, consider again the constrained circuit in Figure 27.5. Assume that the search first branches on the gate g , assigning it to **T**. No truth value is propagated by this. Then assume that the next branching step assigns the gate h to **T**, propagating b and f to **T** and c to **F**. The value of the originally constrained gate j is now justified by the value of h , the value of h is justified by the values of b and f , and so on down to the primary input gates. Therefore, the partial assignment $\{b \mapsto \mathbf{T}, c \mapsto \mathbf{F}\}$ on primary inputs is enough to make the gate j to evaluate to **T**; we can pick an arbitrary value for the unassigned primary input gate a and evaluate all the other gates to obtain a satisfying truth assignment for the constrained circuit. But note that, independent of the value we choose for a , the gate g will evaluate to **F**, not to **T** as assigned in the first branching step of the search. This can happen because, after the second branching step and subsequent propagation, the value of the gate g is in fact unobservable with respect to the original constraints of the circuit. If such unobservability can be detected during the search, the satisfiability of the circuit can sometimes be decided and the search terminated even though the current assignment contains gates that are assigned but not justified. In the running example, for instance, the search can be terminated after the second branching step as g is unobservable. However, if the unobservability of g is *not* detected, the search has to eventually backtrack to the second level, flip the value of the gate h and then find a satisfying assignment in which the gates b and c have the same value so that g can evaluate to **T**.

If branching during the search is restricted to happen in a top-down manner so that branching steps are only made on the gates that are children of currently assigned but unjustified gates, then observability don't cares work similarly to the so-called justification frontier heuristics (see Section 27.3.7.2). However, as the second paragraph above exemplifies, observability don't cares can also be applied when the use of the branching rule is not restricted.

In addition to satisfiability checking, observability don't cares have also been exploited in circuit-based quantified Boolean formula evaluation [GIB09].

Implementation Techniques

In [GGZA01] a CNF-based SAT solver is applied but some information is brought from the circuit level to the CNF level when the circuit-to-CNF translation is applied. Namely, each CNF variable (corresponding to a gate in the circuit) is attached with the list of clauses that describe the functionality of the gate in the CNF formula. After each branching and Boolean constraint propagation, starting

from the variables corresponding to the primary output gates, a sweep over the CNF formula variables is made and the non-don't care variables (gates) as well as the clauses that define their functionality are marked by using the attached clause information on the yet unsatisfied clauses. This marking information is then used to dynamically remove the functionality defining clauses of the don't care variables from the CNF formula.

In [SVDL04] a similar technique is applied but directly at the circuit level. Each gate is marked either as non-don't care (free in the paper) or don't care (lazy in the paper). After each branching and Boolean constraint propagation, the don't care information is recomputed by using the circuit structure and the current truth assignment. The don't care information is exploited by excluding the don't care gates from the set of gates that the SAT solver can branch on. The way the don't care information is computed makes it possible to exploit the “masking of earlier decisions” phenomenon described above.

Computation of don't care information is improved in [TBW04] by using a “don't care watch parent” technique, resembling the watched literal technique applied in Boolean constraint propagation (see Chapter 4 in this handbook and Section 27.3.5.4). The idea is that each gate (excluding primary outputs) is associated with a “don't care watch parent” that is one of its parents. When a gate becomes justified or don't care while being a “don't care watch parent” of one of its children, then one tries to find another parent of the child that is not yet justified or don't care and mark that as the new “don't care watch parent” of the child; if no such parent is found, the child becomes a don't care and the same process is applied recursively. This technique helps especially when a gate has a large number of parents ([TBW04] reports that some circuits they experimented with have 23 parents per gate on average) as it is not necessary to always check whether a gate becomes a don't care when one its parents becomes justified or don't care.

27.3.7. Structure-Based Search Heuristics

Compared to the flat CNF encoding of a problem, the structure preserving Boolean circuit representation offers some natural heuristics for guiding DPLL-based search procedures.

27.3.7.1. Branching on Primary Input Gates Only

In a Boolean circuit encoding of a structural problem typically only a very small portion of the gates are primary input gates. As the values of the primary input gates determine the values of all the other gates, it is sufficient to branch (i.e. to use the explicit cut rule) only on them in DPLL-based backtracking search. Intuitively this helps in reducing the worst case size of the search space from the order of $2^{|\mathcal{G}|}$ to $2^{|\text{inputs}(\mathcal{C})|}$, where $|\text{inputs}(\mathcal{C})| \leq |\mathcal{G}|$ for a circuit \mathcal{C} with a set of gates \mathcal{G} . This has been observed and experimentally evaluated in many papers in the area of SAT e.g. [GMS98, Str00, GMT02, JN08] and automated test pattern generation [Goe81]. According to the experimental evaluations, the effect of primary input restricted branching seems to be highly dependent on the problem class: both positive and negative results have been obtained (see e.g. [GMT02, JN08]).

However, if the best-case complexity is considered, then it can be proven that branching only on primary input gates can actually be very harmful for the solver efficiency. That is, there are families of Boolean circuits such that (i) for primary input restricted branching the smallest possible search space for a DPLL-based algorithm is of super-polynomial size with respect to the size of the circuit, while (ii) there are polynomially sized search spaces when branching is allowed on all gates [JJN05, JJ07].

27.3.7.2. Top-Down Search with Justification Frontiers

An opposite approach to guiding the DPLL-based search is to start from the constrained gates of the circuit and branch downwards in the circuit structure, see e.g. [KPKG02, LWCH03]. That is, branching is performed only on the children of currently assigned but not yet justified gates, trying to justify the value of the gate in question. For instance, if there is an **and**-gate that is assigned to false but has none of its children assigned to false, then the children are potential gates to be branched on (preferably with the value false as it makes the value of the **and**-gate justified). If there are no gates that are assigned but not justified, then the current, potentially partial truth assignment can be extended to a satisfying total truth assignment and the search can be terminated.

A slight variant of the DPLL-style top-down branching scheme is presented e.g. in [KGP01], where the branching is not performed on a gate and its values but on a set of gates. For instance, if there is a binary **and**-gate that is assigned to false but is not yet justified, then two branches are considered: the one in which the “left” child is false, and the other in which the “right” child is false.

In [JJN05, JJ08] it is shown that, for the DPLL-based search procedure (both with and without conflict driven learning), this kind of top-down use of branching can negatively effect the sizes of the minimal search spaces; there are families of circuits for which the minimal search spaces are super-polynomially larger than in the case of allowing unrestricted branching on all gates.

27.3.7.3. Exploiting Signal Correlation

Another interesting way to exploit circuit structure for guiding DPLL-based search is presented in [LWCH03, LWC⁺04]. Two gates, g and g' , are said to be positively (negatively, respectively) correlated if it holds with high probability that their values are the same (the opposite, respectively) in randomly generated consistent truth assignments for the circuit. In fact, g' can also be the constant **F** meaning if a gate g is positively (negatively, respectively) correlated with **F**, then it's value is **F** (**T**, respectively) with high probability. The correlations are approximated by performing a fixed amount of simulations on the circuit: the primary input gates are assigned to random values, the rest of the gates are evaluated, and the correlation relationship is refined according to the simulated values.

The correlations are exploited in two ways. First, whenever a gate g is assigned to a value in the search, the gates that are correlated with it are favored in the branching heuristics and are more likely to be assigned with the value that is against the correlation. The reasoning behind this is that if two gates are correlated, then assigning them values against the correlation is likely to cause the

search to encounter conflicts quickly and thus learn valuable information on the search space by using conflict driven learning.

The second way to exploit correlations is to use them together with the topological structure of the circuit to form smaller sub-problems that are solved in a bottom-up fashion by the DPLL-search. For instance, if a gate g is negatively correlated with the constant \mathbf{F} , then the subproblem consisting of the gate and its descendant gates only is tried to be solved with the constraint that g is assigned to \mathbf{T} . This is likely to cause valuable conflict clauses to be found. After this subproblem, the search will move on to subproblems that are topologically higher and reuse the conflict clauses found while solving the subproblem.

27.3.8. Non-clausal stochastic local search methods

The more structural representation offered by non-CNF formulas and circuits has also been used in many satisfiability algorithms based on stochastic local search (see Chapter 6 in this handbook for clausal stochastic local search methods such as GSAT and Walksat). In many formula-level approaches [Seb94, PTS07, Sta02, SB08], the flips are only applied to the variables appearing in the formula (corresponding to input gates in circuits) while the variable selection is guided by the structure of the formula.

A different approach is taken in [JJN08a, JJN08b], where the circuit-level Walksat-style local search is driven mainly top-down so that the flips are performed on the justification frontier gates and their children. This allows the search to exploit the observability don't cares. To make the search probabilistically approximately complete, i.e., to ensure that a satisfying assignment (if one exists) is eventually found with probability 1 without using restarts, "upward" flips on the justification frontier gates as well as choosing non-greedy justifications for justified gates are also done with some non-zero probability. The justification-frontier approach is modified in [BS10, BJS11] so that the flips can be performed on all unjustified gates, whether in the frontier or below it. In addition, a bounded amount of constraint propagation is performed after each flip.

27.4. Automatic Test Pattern Generation

After producing a chip the functional correctness of the integrated circuit has to be checked. Otherwise products with malfunctions would be delivered to customers which is not acceptable for any company. During this post-production test input stimuli are applied and the correctness of the output response is controlled. These input stimuli are called test patterns. Many algorithms for *Automatic Test Pattern Generation* (ATPG) have been proposed in the last 30 years. But due to the ever increasing design complexity new techniques have to be developed that can cope with today's circuits.

While classical approaches are based on backtracking on the circuit structure, since the early 80s several approaches based on *Boolean Satisfiability* (SAT) have been proposed (see e.g. [SBSV96, MSS97, TGH97, TG99]). In [SBSV96] comparisons to more than 40 other "classical" approaches based on FAN, PODEM and

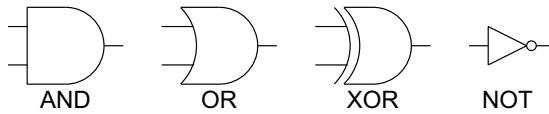


Figure 27.6. Basic gates

the D-algorithm are provided showing the robustness and effectiveness of SAT-based techniques. In contrast to the early work in the 80s, where ATPG was often reduced to PROLOG or logic programming, the ATPG techniques developed by Larrabee [Lar92] and Stephan et al. [SBSV96] used many simplification and learning approaches, like global implications [TGH97]. By this, these algorithms combined the power of SAT with algorithms from the testing domain.

Recently, there is a renewed interest in SAT, and many improvements for proof engines have been proposed. SAT solvers make use of learning and implication procedures (see e.g. [MSS99, MMZ⁺01] and Chapters 3, 4 and 5 of this handbook).

In this section we give an introduction to ATPG. The basic concept and classical ATPG algorithms are briefly reviewed. Then, the formulation as a SAT problem is considered. The transformation of ATPG onto SAT is discussed. Advanced techniques for SAT-based ATPG are explained for the ATPG tool *PASSAT* (PAttern Search using SAT). In contrast to previous SAT approaches, which only considered Boolean values, we study the problem for a multi-valued logic encoding, i.e. *PASSAT* can also consider unknown values and tri-states. The influence of different branching heuristics is studied to tune the SAT solver towards test generation, i.e. variable selection strategies known from SAT and strategies applied in classical ATPG [Goe81, FS83]. Some experimental results are shown to give an impression on the efficiency of the approach.

27.4.1. Preliminaries

This section provides the necessary notions to introduce the ATPG problem. First, fault models are presented. Then, the reduction of a sequential ATPG problem to a combinational problem is explained. Finally, classical ATPG algorithms working on the circuit structure are briefly reviewed. The presentation is kept brief, for further reading we refer to [JG03].

27.4.1.1. Stuck-At Fault Model

The definition of a circuit has been formally introduced at the beginning of this chapter. To simplify the readability, in the following we assume that the circuits consist of gates of type *and*, *or*, *xor* and *not* only (see Figure 27.6). Extending this library to other Boolean gates if necessary is straightforward. Notice again that for gates that represent non-symmetric functions (e.g. multiplexors or tri-state elements) a unique order for the inputs is given by ordering the predecessors of a gate.

After producing a chip the functional correctness of this chip with respect to the Boolean gate-level specification has to be checked. Without this check an

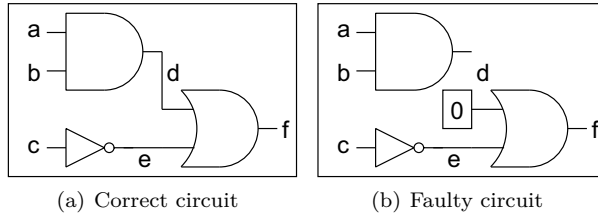


Figure 27.7. Example for the SAFM

erroneous chip would be delivered to customers that may result in a malfunction of the final product. This, of course, is not acceptable. On the other hand a large range of malfunctions is possible due to defects in the material, process variations during production etc. But directly checking for all possible physical defects is not feasible. Therefore an abstraction in terms of a *fault model* is introduced.

The *Stuck-At Fault Model* (SAFM) [BF76] is well-known and widely used in practice. In this fault model a single line is assumed to be stuck at a fixed value instead of depending on the input values. When a line is stuck at the value 0, this is called a *stuck-at-0* (SA0) fault. Analogously, if the line is stuck at the value 1, this is a *stuck-at-1* (SA1) fault.

Example 27.4.1. Consider the circuit shown in Figure 27.7(a). When a SA0 fault is introduced on line *d* the faulty circuit in Figure 27.7(b) is resulting. The output of the and-gate is disconnected and the input of the or-gate constantly assumes the value 0.

Besides the SAFM a number of other fault models have been proposed, e.g. the cellular fault model [Fri73], where the function of a single gate is changed, or the bridging fault model [KP80], where two lines are assumed to settle to a single value. These fault models mainly cover static physical defects like opens or shorts. Dynamic effects are covered by delay fault models. In the path delay fault model [Smi85] a single fault means that a value change along a path from the inputs to the outputs in the circuit does not arrive within the clock-cycle time. Instead of paths the gate delay fault model [HRVD77, SB77] considers the delay at gates.

In the following only the SAFM is considered further due to the high relevance in practical applications. This relevance can be attributed to two observations: the number of faults is in the order of the size of the circuit and fault modeling in the SAFM is relatively easy, i.e. for the static fault model the computational complexity of test pattern generation is lower compared to dynamic fault models.

27.4.1.2. Combinational ATPG

Automatic Test Pattern Generation (ATPG) is the task of calculating a set of test patterns for a given circuit with respect to a fault model. A test pattern for a particular fault is an assignment to the primary inputs of the circuit that leads to different output values depending on the presence of the fault. Calculating the Boolean difference of the faulty and fault free circuit yields all test patterns for a

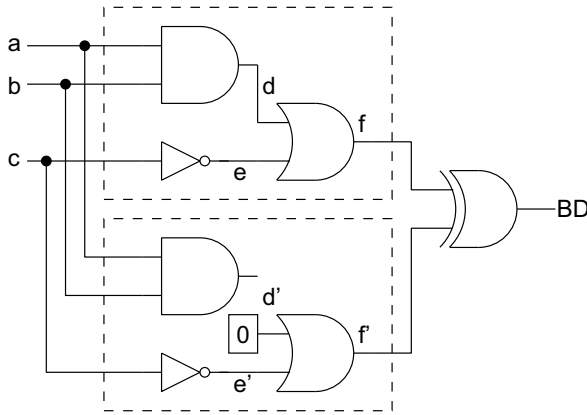


Figure 27.8. Boolean difference of faulty circuit and fault free circuit

particular fault. This construction is similar to a Miter circuit [Bra83] as it can be used for combinational equivalence checking.

Example 27.4.2. Again, consider the SA0 fault in the circuit in Figure 27.7. The input assignment $a = 1, b = 1, c = 1$ leads to the output value $f = 1$ for the correct circuit and to the output value $f = 0$ if the fault is present. Therefore this input assignment is a test pattern for the fault d SA0. The construction to calculate the Boolean difference of the fault free circuit and faulty circuit is shown in Figure 27.8.

When a test pattern exists for a particular fault, this fault is classified as being *testable*. When no test pattern exists, the fault is called *redundant*. The decision problem to classify a fault as redundant or testable is **NP**-complete. The aim is to classify all faults and to create a set of test patterns that contains at least one test pattern for each testable fault.

Generating test patterns for circuits that contain state elements like flip-flops is computationally more difficult, because the state elements can not directly be set to a particular value. Instead the behavior of the circuit over time has to be considered during ATPG. A number of tools have been proposed that directly address this sequential problem, e.g. HITEC [NP91]. But in practice, the resulting model often is too complex to be handled by ATPG tools. Therefore the *full scan mode* is often considered to overcome this problem by connecting all state elements in a scan chain [WA73, EW77]. In the test mode the scan chain combines all state elements into a shift register, in normal operation mode the state elements are driven by the ordinary logic in the circuit. As a result the state elements can be considered as primary inputs and outputs for testing purposes and a combinational problem results.

27.4.1.3. Classical ATPG Algorithms

Classical algorithms for ATPG usually work directly on the circuit structure to solve the ATPG problem for a particular fault. Some of these algorithms are

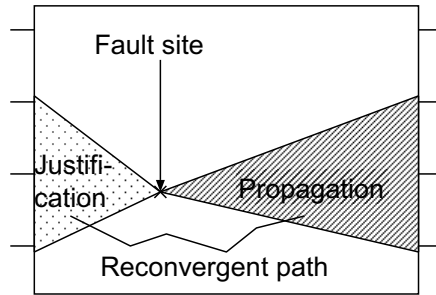


Figure 27.9. Justification and propagation

briefly reviewed in the following. For an in-depth discussion we refer the reader to text books on ATPG, e.g. [JG03].

One of the first complete algorithms dedicated to ATPG was the *D-algorithm* proposed by Roth [Rot66]. The basic ideas of the algorithm can be summarized as follows:

- An error is observed due to differing values at a line in the circuit with or without failure. Such a divergence is denoted by values D or \bar{D} to mark differences 1/0 or 0/1, respectively.
- Instead of Boolean values, the set $\{0, 1, D, \bar{D}\}$ is used to evaluate gates and carry out logical implications by propagating the values.
- A gate that is not on a path between the error and any output never has a D -value.
- A necessary condition for testability is the existence of a path from the error to an output, where all intermediate gates either have a D -value or are not assigned yet. Such a path is called a potential D-chain.
- A gate is on a D-chain, if it is on a path from the error location to an output and all intermediate gates have a D -value.

On this basis an ATPG algorithm can focus on justifying a D -value at the fault site and propagating this D -value to an output as shown in Figure 27.9. The algorithm starts with injecting the D -value at the fault site. Then, this value has to be propagated towards the outputs. For example, to propagate the value D at one input across a 2-input and-gate the other input must have the non-controlling value 1. After reaching an output the search proceeds towards the inputs in the same manner to justify the D -value at the fault site. At some stages in the search decisions are possible. For example, to produce a 0 at the output of an and-gate either one or both inputs can have the value 0. Such a decision may be wrong and may lead to a conflict later on. For example, due to a reconvergence as shown in Figure 27.9 justification may not be possible due to conditions from propagation. In this case, a backtrack search has to be applied. In summary the D-algorithm is confronted with a search space of $O(2^s)$ for a circuit with s signals including inputs, outputs and internal signals.

A number of improvements have been proposed for this basic procedure. PODEM [Goe81] for instance branches only on the values for primary inputs.

This reduces the search space for test pattern generation to $O(2^n)$ for a circuit with n primary inputs. But as a disadvantage time is wasted, if all internal values are implied from a given input pattern that finally does not detect the fault. FAN [FS83] improves upon this problem by branching on stems of fan-out points as well, i.e. all internal points are considered where signals branch. This allows to calculate conditions for a test pattern due to the internal structure of the circuit as well. The branching order and value assignments are determined by heuristics. Moreover, the algorithm keeps track of a justification frontier moving towards the inputs and a propagation frontier moving towards the outputs. Therefore FAN can make the “most important decision” first – based on a heuristic – while the D-algorithm applied a more static order doing only propagation at first and justification afterwards. SOCRATES [STS87] includes the use of global static implications by considering the circuit structure. Based on particular structures in the circuit indirect implications are possible, i.e. implications that are not directly obvious due to assignments at a single gate, but implications that result from functional arguments across several gates. These indirect implications are directly applied during the search process to imply values earlier from partial assignments and, by this, prevent wrong decisions. HANNIBAL [Kun93] further improves this idea. While SOCRATES only uses a predefined set of indirect implications, HANNIBAL learns from the circuit structure in a preprocessing step. For this task recursive learning [KP94] is applied. In principle, recursive learning is complete itself, but too time consuming to be used as a stand alone procedure. Therefore learning is done in a preprocessing step. During this step the effect of value assignments is calculated and the resulting implications are learned. These implications are stored for the following run of the search procedure. In HANNIBAL the FAN algorithm was used to realize this search step.

These algorithms only address the problem of generating a test pattern for a single fault. In the professional application such basic ATPG procedures are complemented by a preprocessing step and a postprocessing step. In the preprocessing phase easy-to-detect faults are classified in order to save run time afterwards. Postprocessing concentrates on test pattern compaction, i.e. reducing the number of test patterns in the test set.

27.4.2. Boolean Satisfiability

Before introducing an ATPG engine based on *Boolean Satisfiability* the standard conversion of a given circuit into a set of clauses for a SAT solver is briefly reviewed to make the chapter self-contained. For a more detailed introduction see Chapter 2 in this handbook.

27.4.2.1. Circuit to CNF Conversion

A SAT solver can be applied as a powerful black-box engine to solve a problem. In this case transforming the problem instance into a SAT instance and the SAT solution into a solution for the original problem is crucial. In particular, for SAT-based ATPG one step is the transformation of the circuit into a CNF. For example, in [Lar92] the basic procedure has been presented.

Table 27.1. Transformation of an and-gate into CNF

(a) Truth-table			(b) Clauses	(c) Minimized
<i>a</i>	<i>b</i>	<i>c</i>	$c \leftrightarrow a \wedge b$	
0	0	0	1	
0	0	1	0	$a \vee b \vee \bar{c}$
0	1	0	1	
0	1	1	0	$a \vee \bar{b} \vee \bar{c}$
1	0	0	1	
1	0	1	0	$\bar{a} \vee b \vee \bar{c}$
1	1	0	0	$\bar{a} \vee \bar{b} \vee c$
1	1	1	1	

The transformation of a single and-gate into a set of clauses is shown in Table 27.1. The goal is to create a CNF that models an and-gate, i.e. a CNF that is only satisfied for assignments that may occur for an and-gate. For an and-gate with two inputs a and b , the output c must always be equal to $a \wedge b$. The truth-table for this CNF formula is shown in Table 27.1(a). From the truth-table a CNF formula is generated by extracting one clause for each assignment where the formula evaluates to 0. These clauses are shown in Table 27.1(b). This CNF representation is not minimal and can therefore be reduced by two-level logic minimization, e.g. using SIS [SSL⁺92]. The clauses in Table 27.1(c) are the final result.

Now, the generation of the CNF for a complete circuit is straightforward. For each gate, clauses are generated according to the type of the gate. The output variables and input variables of a gate and its successors are identical and therefore establish the overall circuit structure within the CNF. This approach gives the well-known Tseitin encoding of a circuit discussed in Chapter 2 in this handbook.

Example 27.4.3. Consider the circuit shown in Figure 27.7(a). This circuit is translated into the following CNF formula:

$$\begin{aligned}
 & \underbrace{(a \vee \bar{d}) \wedge (b \vee \bar{d}) \wedge (\bar{a} \vee \bar{b} \vee d)}_{d \leftrightarrow a \wedge b} \\
 & \wedge \underbrace{(c \vee e) \wedge (\bar{c} \vee \bar{e})}_{e \leftrightarrow \bar{c}} \\
 & \wedge \underbrace{(\bar{d} \vee f) \wedge (\bar{e} \vee f) \wedge (d \vee e \vee \bar{f})}_{f \leftrightarrow d \vee e}
 \end{aligned}$$

An advantage of this transformation is the linear size complexity. Given a circuit where n is the sum of the numbers of inputs, outputs and gates, the number of variables in the SAT instance is also n and the number of clauses is in $O(n)$.

A disadvantage is the loss of structural information. Only a set of clauses is presented to the SAT solver. Information about predecessors and successors of a gate is lost and is not used during the SAT search. But as will be shown later this information can be partially recovered by introducing additional constraints into the SAT instance.

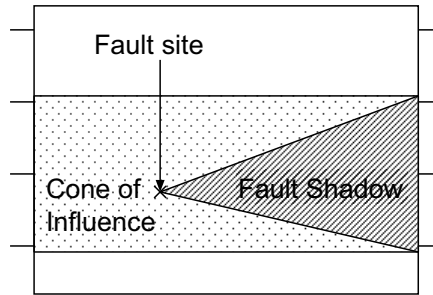


Figure 27.10. Influenced circuit parts

27.4.3. SAT-based ATPG

In this section SAT-based ATPG is explained in detail. The basic problem transformation is presented at first. Then, an improvement of this basic transformation by exploiting structural information is shown. This is enhanced for the practical application to multi-valued modelling of circuits.

27.4.3.1. Basic Problem Transformation

In the following we describe a SAT-based ATPG tool that works on a SAT instance in CNF. It would also be possible to use a circuit SAT solver, but by experiments so far the CNF-based solvers turned out to be more efficient for this application.

The transformation is formulated for a single fault. Then, the process iterates over all faults to generate a complete test set. Alternatively, the SAT-based engine can be integrated within an ATPG tool that uses other engines as well.

First, the fault site is located in the circuit. Then, the parts of the circuit that are influenced by the fault site are calculated as shown in Figure 27.10. From the fault site towards the outputs all gates in the transitive fan-out, i.e. all gates that can structurally be reached from the faulty line, are influenced, this is also called the *fault shadow*. Then, the cone of influence of the outputs in the fault shadow is calculated. These gates have to be considered when creating the ATPG instance. Analogously to the construction of the Boolean difference shown in Figure 27.8 a fault free model and a faulty model of the circuit are joined into a single SAT instance to calculate the Boolean difference between both versions. All gates not contained in the transitive fan-out of the fault site have the same behavior in both versions. Therefore only the fault shadow is duplicated as shown in Figure 27.11. Then, the Boolean difference of corresponding outputs is calculated. At least one of these Boolean differences must assume the value 1 to obtain a test pattern for the fault. In Figure 27.11 this corresponds to constraining the output of the or-gate to the value 1. Now, the complete ATPG instance is formulated in terms of a circuit and can be transformed into a CNF. If the SAT solver returns a satisfying assignment this directly determines the values for the primary inputs to test the fault. If the SAT solver returns unsatisfiable, the considered fault is redundant.

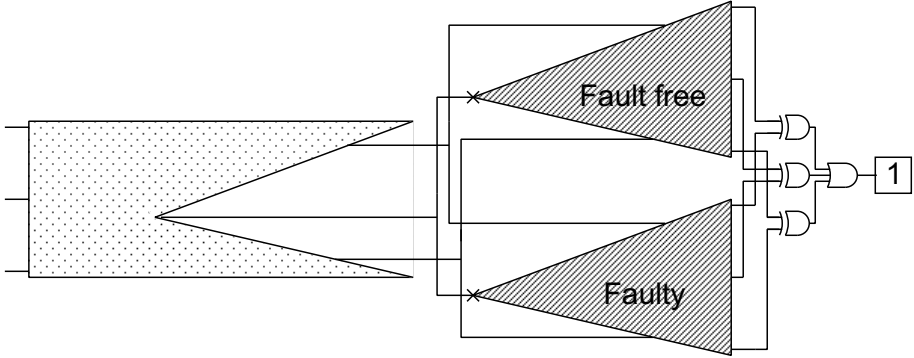


Figure 27.11. Influenced circuit parts

27.4.3.2. Structural Information

As explained earlier most of the structural information is lost during the transformation of the original problem into CNF. But this can be recovered by additional constraints. This has been suggested for the SAT-based test pattern generator TEGUS [SBSV96]. Improvements on the justification and propagation have been proposed in [TGH97, TG99].

In particular, the observations from the D-algorithm as explained in Section 27.4.1.3 are made explicit in the CNF. Now, three variables are used for each gate g :

- g_f denotes the value in the faulty circuit.
- g_c denotes the value in the correct circuit.
- $g_d = 1$, iff g is on a D-chain.

This notation allows to introduce additional implications into the CNF:

- If g is on a D-chain, the values in the faulty and the correct circuit are different: $g_d \rightarrow (g_f \neq g_c)$.
- If g is on a D-chain, at least one successor of g must be on the D-chain as well: Let $h^i, 1 \leq i \leq q$ be the successors of g , then $g_d \rightarrow \bigvee_{i=1}^q h_d^i$.
- If a gate g is not on a D-chain, the values in the faulty and the correct circuit are identical: $\bar{g}_d \rightarrow (g_f = g_c)$.

Without these implications the fault shadow in the fault free version and the faulty version of the circuit are only connected via the variables on the cut to the shared portions of the circuit. In contrast the g_d variables establish direct links between these structures. Therefore implications are possible even when the cut variables are not assigned yet. Moreover, the information about successors of a gate and the notion of D-chains are directly encoded in the SAT instance.

27.4.3.3. Encoding

In this section the techniques to handle non-Boolean values are discussed. These techniques have been proposed for the tool PASSAT [SFD⁺05, DEF⁺08]. First,

the use of multi-valued logic during ATPG is motivated and the four-valued logic is introduced. Then, different possibilities to encode the multi-valued problem in a Boolean SAT instance are discussed.

27.4.3.3.1. Four-Valued Logic For practical purposes it is not sufficient to consider only the Boolean values 0 and 1 during test pattern generation as it has been done in earlier approaches (e.g. [SBSV96]). This has mainly two reasons.

Firstly, circuits usually have tri-state elements. Therefore, besides the basic gates already shown in Figure 27.6 also tri-state elements may occur in a circuit. These are used if a single signal is driven by multiple sources. These gates behave as follows:

- **busdriver**, Inputs: a, b , Output: c
Function: $c = \begin{cases} Z, & \text{if } a = 0 \\ b, & \text{if } a = 1 \end{cases}$
- **bus0**, Inputs: a_1, \dots, a_n , Output c
Function: $c = \begin{cases} 0, & \text{if } a_1 = Z, \dots, a_n = Z \text{ or } \exists i \in \{1, \dots, n\} a_i = 0 \\ 1, & \exists i \in \{1, \dots, n\} a_i = 1 \end{cases}$
Note, that the output value is not defined, if there are two inputs with the opposite values 0 and 1.
- **bus1** behaves as bus0, but assumes the value 1 if not being driven.
- **bus** behaves as bus0 and bus1, but assumes the value Z if not being driven.

From a modeling point of view the tri-state elements could be transformed into a Boolean structure with the same functionality, e.g. by inserting multiplexers. But during test pattern generation additional constraints apply to signals driven by tri-state elements. For example, no two drivers must drive the signal with opposite values or if all drivers are in the high impedance state the driven signal has an unknown value. The value Z is used to properly model these constraints and the transition function of tri-state elements.

Environment constraints that apply to a circuit are another problem. Usually the circuit is embedded in a larger environment. As a result some inputs of the circuit may not be controllable. Thus, the value of such a non-controllable input is assumed to be unknown during ATPG. The logic value U is used to model this situation. This has to be encoded explicitly in the SAT instance, because otherwise the SAT solver would also assign Boolean values to non-controllable inputs.

Therefore a four-valued logic over $\{0, 1, Z, U\}$ is considered in PASSAT.

27.4.3.3.2. Boolean Encoding The multi-valued ATPG problem has to be transformed into a Boolean problem to use a modern Boolean SAT solver on the four-valued logic. Therefore each signal of the circuit is encoded by two Boolean variables. One encoding out of the $4! = 24$ mappings of four values onto two Boolean values has to be chosen. The chosen encoding determines which clauses are needed to model particular gates. This, in turn, influences the size of the resulting SAT instance and the efficiency of the SAT search.

All possible encodings are summarized in Tables 27.2(a)-27.2(c). The two Boolean variables are denoted by x and \bar{x} , the letters a and b are placeholders for

Table 27.2. Boolean encodings

(a) Set 1

s	x	\bar{x}
0	a	b
1	a	\bar{b}
U	\bar{a}	b
Z	\bar{a}	\bar{b}

(b) Set 2

s	x	\bar{x}
0	a	b
1	a	\bar{b}
U	\bar{a}	\bar{b}
Z	\bar{a}	b

(c) Set 3

s	x	\bar{x}
0	a	b
1	\bar{a}	\bar{b}
U	\bar{a}	b
Z	a	\bar{b}

(d) Example: Set 1,
 $a = 0, b = 0, x = c_s$

s	c_s	c_s^*
0	0	0
1	0	1
U	1	0
Z	1	1

Boolean values. The following notations define the interpretation of the tables more formally:

- A signal s is encoded by the two Boolean variables c_s and c_s^* .
- $x \in \{c_s, c_s^*\}, \bar{x} \in \{c_s, c_s^*\} \setminus \{x\}$
- $a \in \{0, 1\}, \bar{a} \in \{0, 1\} \setminus \{a\}$
- $b \in \{0, 1\}, \bar{b} \in \{0, 1\} \setminus \{b\}$

Example 27.4.4. Consider Set 1 as defined in Table 27.2(a) and the following assignment: $a = 0, b = 0, x = c_s$. Then, the encoding in Table 27.2(d) results.

Thus, a particular encoding is determined by choosing values for a, b and x . Each table defines a set of eight encodings.

Note, that for encodings in Set 1 or Set 2 one Boolean variable is sufficient to decide, if the value of s is in the Boolean domain, i.e. in $\{0, 1\}$, or in the non-Boolean domain, i.e. in $\{U, Z\}$. In contrast encodings in Set 3 do not have this property. This observation will be important when the efficiency of a particular encoding for SAT solving is determined.

27.4.3.3.3. Transformation to SAT Instance The clauses to model a particular gate type can be determined if a particular encoding and the truth-table of the gate are given. This is done analogously to the procedure in Section 27.4.2.1. The set of clauses can be reduced by two-level logic-optimization. Again, the tool ESPRESSO contained in SIS [SSL⁺92] was used for this purpose. For the small number of clauses for the basic gate types ESPRESSO is capable of calculating a minimal representation. The following example illustrates this flow.

Example 27.4.5. Table 27.3(a) shows the truth-table of an and-gate $s = t \wedge u$ over $\{0, 1, Z, U\}$. The truth-table is mapped onto the Boolean domain using the encoding from Example 27.4.4. The encoded truth-table is shown in Table 27.3(b) (for compactness the notation “ $\neq 0$ ” is used to denote that at least one of two variables must be different from 0; “-” denotes “don’t care”). A CNF is extracted from this truth-table and optimized by ESPRESSO.

Statistics for each possible encoding are presented in Table 27.4. For each gate type the number of clauses needed to model the gate are given. Besides the well-known Boolean gates (and, or, ...) also the non-Boolean gates *busdriver*, *bus0* and *bus1* are considered. The last column *All* in the table gives the sum of the numbers of clauses for all gate types.

Table 27.3. and-gate over $\{0, 1, Z, U\}$

(a) 4-valued			(b) Encoded					
t	u	s	c_t	c_t^*	c_u	c_u^*	c_s	c_s^*
0	–	0	0	0	–	–	0	0
–	0	0	–	–	0	0	0	0
1	1	1	0	1	0	1	0	1
U	$\neq 0$	U	1	0	$\neq 0$	0	1	0
Z	$\neq 0$	U	1	1	$\neq 0$	0	1	0
$\neq 0$	U	U	$\neq 0$	0	1	0	1	0
$\neq 0$	Z	U	$\neq 0$	0	1	1	1	0

Table 27.4. Number of clauses for each encoding

Set	nand	nor	and	bus	bus0	bus1	busdriver	xor	not	or	All
1	8	9	9	10	11	10	9	5	5	8	100
2	9	8	8	10	10	11	9	5	5	9	100
3	11	11	11	8	9	9	11	5	6	11	108

Table 27.5. Number of gates for each type

circ.	IN	OUT	FANO.	not	and	nand	or	nor	bus	busdriver
p44k	2356	2232	6845	16869	12365	528	5484	1128	0	0
p88k	4712	4565	14560	20913	27643	2838	16941	5883	144	268
p177k	11273	11031	33605	48582	49911	5707	30933	5962	0	560

All encodings of a given set lead to clauses that are isomorphic to each other. By mapping the polarity of literals and the choice of variables the other encodings of the set are retrieved. Particularly, Boolean gates are modeled efficiently by encodings from Set 1 and Set 2. The sum of clauses needed for all gates is equal for both sets. The difference is that, for example, the encodings of one set are more efficient for **nand**-gates, while the encodings of the other set are more efficient for **nor**-gates. Both gate types occur with a similar frequency in the industrial benchmarks as shown in Table 27.5. The same observation is true for the other gates where the efficiency of the encodings differs. Therefore no significant trade-off for the encodings occurs on the benchmarks.

In contrast more clauses are needed to model Boolean gates if an encoding of Set 3 is used. At the same time this encoding is more efficient for non-Boolean gates. In most circuits the number of non-Boolean gates is usually much smaller than the number of Boolean gates. Therefore more compact SAT instances will result if an encoding from Set 1 or Set 2 is used. The behavior of the SAT solver does not necessarily depend on the size of the SAT instance, but if the same problem is encoded in a much smaller instance also better performance of the SAT solver can be expected. These hypotheses are strengthened by experimental results.

27.4.4. Variable Selection

A SAT solver traverses the search space by a backtracking scheme. While BCP and conflict analysis have greatly improved the speed of SAT solvers, the variable selection strategy is crucial for an efficient traversal of the search space. But no

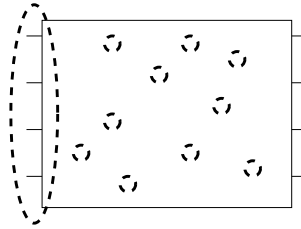


Figure 27.12. Decision variables in dedicated variable selection strategies

general way to choose the best variable is known, as the decision about satisfiability of a given CNF formula is **NP**-complete [Coo71]. Therefore SAT solvers have sophisticated heuristics to select variables (see Chapters 4, 5, and 8 in the handbook). Usually the heuristic accumulates some statistics about the CNF formula during run time of the SAT solver to use this data as the basis for decisions. This leads to a trade-off between the quality of a decision and the overhead needed to update the statistics. Also, the quality of a given heuristic often depends on the problem domain. The default variable selection strategy applied by ZCHAFF is the quite robust VSIDS strategy (see Chapter 4 in the handbook).

Decisions based on variable selection also occur in classical test pattern generation. Here, usually structural methods are employed to determine a good choice for the next selection. Besides the default variable selection strategy from ZCHAFF the tool PASSAT provides two strategies similar to strategies known from classical ATPG: selecting primary inputs only or selecting fan-out points only, i.e. all internal points are considered where signals branch.

Making decisions on primary inputs only was the improvement of PODEM [Goe81] upon the D-algorithm. Any other internal value can be implied from the primary inputs. This yields a reduction of the search space and motivates to apply the same strategy for SAT-based test pattern generation as well. For SAT solving this is done by restricting the variable selection of the SAT solver to those variables corresponding to primary inputs or state bits of the circuit. Within these variables the VSIDS strategy is applied to benefit from the feedback of conflict analysis and current position in the search space. Figure 27.12 visualizes this strategy. Only the variables in the dashed oval are allowed for selection.

Restricting the variable selection to fan-out gates only has been proposed in FAN [FS83] for the first time. Again, the idea is to restrict the search space while getting a large number of implications from a single decision. Conflicts resulting from a decision are often due to a small region within the circuit. PASSAT applies the VSIDS strategy to select fan-out gates or primary inputs. In Figure 27.12 this corresponds to the areas in the dashed oval and in the circles within the circuit. Including primary inputs in this selection strategy is crucial for the SAT procedure because this ensures that all other variable assignments can be implied from decision variables. This is necessary because otherwise conflicts may remain undetected.

The experiments in Section 27.4.6 show that some heuristics are quite robust, i.e. can classify all faults, while others are fast for some faults but abort on other

faults. Therefore an iterative approach turned out to be most effective:

1. One strategy is run with a given time out.
2. If the first strategy does not yield a test pattern a second, more robust, strategy is applied.

Experimental results show that this approach ensures fast test pattern generation where possible, while a more sophisticated search is done for the remaining faults. For the experiments selecting only inputs was used as the first strategy, and selecting any variable as the second.

27.4.5. Experimental Results

In this section we report experimental results to validate the efficiency of particular techniques presented above by discussing the two SAT-based approaches TEGUS and PASSAT. The latter tool is based on the SAT solver ZCHAFF [MMZ⁺01]³. First, PASSAT is compared to previous FAN-based and SAT-based approaches for ATPG. Then, the influence of the encoding is studied. The variable selection strategies are evaluated in another experiment. Finally, results for industrial benchmarks are presented.

27.4.5.1. ISCAS Benchmarks

First we study the run time behavior of the two SAT-based approaches for some of the benchmarks from ISCAS89. (More detailed experimental studies and more information on the parameter settings can be found in original papers.) To demonstrate the quality of SAT-based approaches a comparison to an improved version of Atalanta, that is based on the FAN algorithm [FS83], is given⁴. Atalanta was used to also generate test patterns for each fault. The backtrack limit was set to 10. The results are shown in Table 27.6. The first column gives the name of the benchmark. Then the run time is given in CPU seconds for each approach. Run times for Atalanta with fault simulation and without fault simulation are given in columns *fs* and *no fs*, respectively. (For more details on fault simulation see [JG03].) On circuits “s35932” and “s38584.1” Atalanta gave no results, when fault simulation was disabled. For TEGUS and PASSAT the run time to generate the SAT instance and the time for SAT-solving are separately given in Columns *Eqn* and *SAT*, respectively.

Both SAT approaches are significantly faster than the classical FAN-based algorithm and solve all benchmarks in nearly no time. No fault simulation has been applied in the SAT approaches, therefore the run times should be compared to Atalanta without fault simulation. Especially for large circuits the SAT approaches show a run time improvement of several orders of magnitude. But even when fault simulation is enabled in Atalanta the SAT approaches are faster by up to more than two orders of magnitude (see e.g. “s35932” and “s38584.1”).

³More recent experimental results can be found in [DEF⁺08], where MiniSAT was used as the underlying solver.

⁴Atalanta is available as public domain software from <http://www.ee.vt.edu/~ha/cadtools/cadtools.html>.

Table 27.6. Results for the Boolean circuit model

Circ.	Atalanta		TEGUS		PASSAT	
	fs	no fs	Eqn	SAT	Eqn	SAT
s1494	0.08	0.37	0.01	0.00	0.06	0.01
s5378	1.70	18.37	0.03	0.02	0.37	0.06
s9234.1	18.63	83.90	0.14	0.39	3.06	0.47
s13207.1	18.63	127.40	0.29	0.16	3.03	0.61
s15850.1	27.13	204.27	0.68	0.76	7.66	1.52
s35932	87.40	-	0.47	0.09	2.68	0.28
s38417	131.77	1624.78	0.52	0.24	3.56	0.65
s38584.1	86.30	-	0.63	0.14	4.09	0.75

Table 27.7. Results for the 4-valued circuit model

Circ.	TEGUS					PASSAT				
	Ct	Red	Ab	Eqn	SAT	Ct	Red	Ab	Eqn	SAT
s1494	1230	12	7	0.05	0.89	1237	12	0	1.23	0.55
s5378	2807	20	46	0.37	2.69	2848	25	0	3.97	3.34
s9234.1	2500	75	132	0.66	6.57	2610	97	0	7.99	2.43
s13207.1	5750	181	685	2.56	107	6362	269	0	45.4	41.4
s15850.1	7666	245	539	5.33	55.4	8116	338	0	63.4	21.7
s35932	26453	3968	4	6.61	8.48	26457	3968	0	31.2	32.7
s38417	20212	149	712	9.22	35.8	20884	189	0	79.7	65.2
s38584.1	24680	1079	857	7.37	27.3	25210	1834	0	40.9	15.1

Considering only the SAT approaches, TEGUS is faster for these simple cases. Here test patterns for all faults are generated. In this scenario the speed-up gained by PASSAT for difficult faults (see below) is overruled by the overhead for sophisticated variable selection and conflict analysis. Furthermore the simple two-valued encoding is used in these experiments. Thus, the 4-valued encoding used in PASSAT only generated overhead. This can clearly be seen when looking at the times needed for the generation of the instances. The situation changes, if more complex circuits or tri-states are considered (see below).

27.4.5.2. Encoding

Again, we study the benchmarks from ISCAS89 and consider single stuck-at faults. But this time we use the 4-valued encoding allowing tri-states and unknown values as discussed in Section 27.4.3.3. The results are shown in Table 27.7. Additionally, the numbers of testable, redundant and aborted faults are shown in columns *Ct*, *Red* and *Ab*, respectively⁵. As can be seen, TEGUS now has a large number of aborted faults. The increase in run time compared to the Boolean encoding in Table 27.6 is due to the increased size of the SAT instance as explained in Section 27.4.3.3 and due to more decisions that are necessary in this case.

⁵Notice that the numbers might vary slightly to numbers reported in the literature due to the underlying AND/INVERTER graph structure.

Table 27.8. PASSAT with different variable selection strategies

Circ.	Default			Fan-out			Inputs			Input, Default		
	Ab	Eqn	SAT	Ab	Eqn	SAT	Ab	Eqn	SAT	Ab	Eqn	SAT
s1494	0	1.30	0.51	0	1.30	0.59	0	1.28	0.54	0	1.29	0.55
s5378	0	4.10	3.26	0	4.65	2.94	0	4.47	1.24	0	4.17	1.48
s9234.1	0	7.73	5.43	0	8.06	5.60	0	7.90	2.25	0	8.19	2.01
s13207.1	0	31.60	92.90	30	33.80	100.00	19	32.00	40.00	0	55.40	39.90
s15850.1	0	62.90	79.70	0	67.40	72.20	0	63.50	20.50	0	63.90	20.70
s35932	0	31.90	32.60	0	66.00	11.60	0	31.40	5.32	0	32.00	5.34
s38417	0	80.70	64.60	0	100.00	66.30	0	82.20	31.80	0	82.60	32.50
s38584.1	0	40.80	26.30	0	68.10	27.50	0	40.50	14.10	0	40.80	14.30

27.4.6. Variable Selection

Next, we study the influence of the different variable selection strategies introduced in Section 27.4.4. The 4-valued circuit model is considered. Table 27.8 shows the results for the default strategy known from ZCHAFF, using fan-out gates only, using inputs only, and the iterative strategy. For the iterative strategy a time out of 1 second while selecting only inputs was allowed per fault, then the default strategy was applied. In case of the default strategy no time out was used, for the remaining strategies 2 seconds were used to guarantee that a single fault does not block the test generation for remaining faults. The table shows that the default strategy is the most robust strategy among the non-iterative strategies. No abortions occur for this heuristic for the ISCAS benchmarks considered. The selection of fan-outs only yields improvements in some cases (e.g. “s35932”), but can not classify all faults in several cases. The reason here is that the reduction of the search space does not compensate for the weak influence of conflict analysis on further decisions. The search space is more drastically pruned, when only inputs are allowed for selection. This yields a large speed up of up to 6 times in case of “s35932”. On the other hand difficult faults are often aborted, because internal conflicts during SAT-solving are detected too late. Therefore the iterative strategy was applied to retrieve a fast solution where possible, while using the more sophisticated VSIDS strategy on all variables, where a fast solution was not possible. As a result the iterative strategy does not lead to any aborted faults. In most cases even a remarkable speed-up compared to the default strategy is gained. Only for a few cases a penalty occurs. But the improvement often is significant, e.g. less than 20% are needed in case of “s35932”. Especially for the larger benchmarks significant run time improvements were achieved.

27.4.6.1. Industrial Circuits

Finally, two industrial circuits from Philips Semiconductors are considered, i.e. large circuits that also contain buses. We gave a 20 second CPU time limit to the SAT solver. The results are given in Table 27.9. As can be seen for these circuits only very few faults remain unclassified, i.e. only 204 for circuit “p565k” that consists of more than 565k gates. For the same benchmark TEGUS gives up on more than 340,000 faults (!) demonstrating how hard the circuit is to test.

Table 27.9. PASSAT for industrial benchmarks

Circ.	Ct	Red	Ab	Eqn	SAT
p88k	127,084	2,354	14	12,186	9,584
p565k	1,203,690	26,426	204	5,291	8,155

27.5. Conclusions

In this chapter we have surveyed methods for solving non-clausal SAT problems and automatic test pattern generation (ATPG) techniques. For non-clausal SAT we have focused on Boolean circuit satisfiability checking, that is, on the variant of propositional satisfiability problem where the input formula is given as a Boolean circuit and on techniques which work directly with circuit representation. We have shown how the highly successful DPLL method for checking satisfiability of a formula in CNF (see Chapter 3 in this handbook) can be generalized to work directly with circuits and how to incorporate, for instance, efficient circuit level Boolean constraint propagation techniques, conflict driven learning, don't care values, and search heuristics exploiting circuit structure.

For ATPG we have reviewed basic concepts and classical algorithms and explained how to formulate ATPG as a SAT problem. We have also presented an efficient approach for SAT-based test pattern generation aiming at circuits that include tri-state elements. Due to recent advances in SAT solving the approach is more robust than previous ones.

Acknowledgments

The first author has been supported by the BMBF project MAYA under contract number 01M3172B and by DFG grant DR 287/15-1, The second author has been supported by the Academy of Finland grant 112016 and the third author by the Academy of Finland grants 211025 and 122399 which is gratefully acknowledged.

References

- [ABE00] P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In *Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference, TACAS 2000, Proceedings*, pages 411–425, 2000.
- [AH02] H. R. Andersen and H. Hulgaard. Boolean expression diagrams. *Information and Computation*, 179(2):194–212, 2002.
- [BB03] O. Bailleux and Y. Boufkhad. Efficient CNF encoding of boolean cardinality constraints. In F. Rossi, editor, *Principles and Practice of Constraint Programming – CP 2003*, volume 2833 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2003.
- [BB04] O. Bailleux and Y. Boufkhad. Full CNF encoding: The counting constraints case. In *SAT 2004; The Seventh International Conference on*

- Theory and Applications of Satisfiability Testing*, 2004. Online proceedings at <http://www.satisfiability.org/SAT04/programme/index.html>.
- [BdlT92] T. Boy de la Tour. An optimality result for clause form translation. *Journal of Symbolic Computation*, 14(4):283–301, 1992.
- [Bes07] C. Bessiere, editor. *Principles and Practice of Constraint Programming – CP 2007*, volume 4741 of *Lecture Notes in Computer Science*. Springer, 2007.
- [BF76] M. A. Breuer and A. D. Friedman. *Diagnosis & reliable design of digital systems*. Computer Science Press, 1976.
- [BJS11] A. Belov, M. Jarvisalo, and Z. Stachniak. Depth-driven circuit-level stochastic local search for SAT. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, pages 504–509. IJCAI/AAAI, 2011.
- [Bor97] A. Borålv. The industrial success of verification tools based on stålmarck’s method. In *Computer Aided Verification, 9th International Conference, CAV ’97, Haifa, Israel, June 22–25, 1997, Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 7–10, 1997.
- [Bra83] D. Brand. Redundancy and don’t cares in logic synthesis. *IEEE Transactions on Computers*, 32(10):947–952, 1983.
- [Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
- [Bry92] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
- [BS10] A. Belov and Z. Stachniak. Improved local search for circuit satisfiability. In O. Strichman and S. Szeider, editors, *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing (SAT 2010)*, volume 6175 of *Lecture Notes in Computer Science*, pages 293–299. Springer, 2010.
- [CA93] J. M. Crawford and L. D. Auton. Experimental results on the crossover point in satisfiability problems. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI’93)*, pages 21–27. The AAAI Press/The MIT Press, 1993.
- [Coo71] S. A. Cook. The complexity of theorem proving procedures. In *3. ACM Symposium on Theory of Computing*, pages 151–158, 1971.
- [D’A99] M. D’Agostino. Tableau methods for classical propositional logic. In *Handbook of Tableau Methods*, pages 45–123. Kluwer Academic Publishers, 1999.
- [DEF⁺08] R. Drechsler, S. Eggersglüß, G. Fey, A. Glowatz, F. Hapke, J. Schöffel, and D. Tille. On acceleration of SAT-based ATPG for industrial designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1329–1333, July 2008.
- [DG84] W. F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *The Journal of Logic Programming*, 1(3):267–284, 1984.
- [DGHE99] M. D’Agostino, D. M. Gabbay, R. Hähnle, and J. P. (Eds). *Handbook*

- of *Tableau Methods*. Kluwer Academic Publishers, 1999.
- [DM94] M. D’Agostino and M. Mondadori. The taming of the cut: Classical refutations with analytic cut. *Journal of Logic and Computation*, 4(3):285–319, 1994.
- [EMS07] N. Eén, A. Mishchenko, and N. Sörensson. Applying logic synthesis for speeding up SAT. In *Theory and Applications of Satisfiability Testing — SAT 2007*, volume 4501 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2007.
- [EW77] E. B. Eichelberger and T. W. Williams. A logic design structure for LSI testability. In *Design Automation Conf.*, pages 462–468, 1977.
- [Fri73] A. D. Friedman. Easily testable iterative systems. *IEEE Transactions on Computers*, 22:1061–1064, 1973.
- [FS83] H. Fujiwara and T. Shimono. On the acceleration of test generation algorithms. *IEEE Transactions on Computers*, 32:1137–1144, 1983.
- [GGZA01] A. Gupta, A. Gupta, Z. Zhang, and P. Ashar. Dynamic detection and removal of inactive clauses in SAT with application in image computation. In *Proceedings of the 38th Design Automation Conference (DAC 2001)*, pages 536–541. ACM, 2001.
- [GIB09] A. Goultiaeva, V. Iverson, and F. Bacchus. Beyond CNF: A circuit-based QBF solver. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT 2009)*, volume 5584 of *Lecture Notes in Computer Science*, pages 412–426. Springer, 2009.
- [GMS98] E. Giunchiglia, A. Massarotto, and R. Sebastiani. Act, and the rest will follow: Exploiting determinism in planning as satisfiability. In *AAAI*, pages 948–953. AAAI Press, 1998.
- [GMT02] E. Giunchiglia, M. Maratea, and A. Tacchella. Dependent and independent variables in propositional satisfiability. In S. Flesca, S. Greco, N. Leone, and G. Ianni, editors, *Logics in Artificial Intelligence, JELIA 2002*, volume 2424 of *Lecture Notes in Artificial Intelligence*. Springer, 2002.
- [Goe81] P. Goel. An implicit enumeration algorithm to generate tests for combinational logic circuits. *IEEE Transactions on Computers*, c-30(3):215–222, March 1981.
- [GZA⁺02] M. K. Ganai, L. Zhang, P. Ashar, A. Gupta, and S. Malik. Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver. In *Proceedings of the 39th Design Automation Conference, DAC 2002*, pages 747–750. ACM, 2002.
- [Häh01] R. Hähnle. Tableaux and related methods. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 100–178. Elsevier and MIT Press, 2001.
- [HRVD77] E. R. Hsieh, R. A. Rasmussen, L. J. Vidunas, and W. T. Davis. Delay test generation. In *Design Automation Conf.*, pages 486–491, 1977.
- [JBH12] M. Järvisalo, A. Biere, and M. Heule. Simulating circuit-level simplifications on CNF. *Journal of Automated Reasoning*, 49(4):583–619, 2012.
- [JG03] N. Jha and S. Gupta. *Testing of Digital Systems*. Cambridge Univer-

sity Press, 2003.

- [JJ07] M. Järvisalo and T. Junttila. Limitations of restricted branching in clause learning. In Bessiere [Bes07], pages 348–363.
- [JJ08] M. Järvisalo and T. Junttila. On the power of top-down branching heuristics. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI-08)*, pages 304–309. AAAI Press, 2008.
- [JJN05] M. Järvisalo, T. Junttila, and I. Niemelä. Unrestricted vs restricted cut in a tableau method for Boolean circuits. *Annals of Mathematics and Artificial Intelligence*, 44(4):373–399, 2005.
- [JJN08a] M. Järvisalo, T. Junttila, and I. Niemelä. Justification-based local search with adaptive noise strategies. In *Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2008)*, volume 5330 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2008.
- [JJN08b] M. Järvisalo, T. Junttila, and I. Niemelä. Justification-based non-clausal local search for SAT. In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI 2008)*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 535–539. IOS Press, 2008.
- [JN00] T. A. Junttila and I. Niemelä. Towards an efficient tableau method for Boolean circuit satisfiability checking. In *Computational Logic – CL 2000; First International Conference*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 553–567. Springer, 2000.
- [JN08] M. Järvisalo and I. Niemelä. The effect of structural branching on the efficiency of clause learning SAT solving: An experimental study. *Journal of Algorithms*, 63(1–3):90–113, 2008.
- [JS05] P. Jackson and D. Sheridan. Clause form conversions for boolean circuits. In H. H. Hoos and D. G. Mitchell, editors, *SAT 2004*, volume 3542 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2005.
- [KGP01] A. Kuehlmann, M. K. Ganai, and V. Paruthi. Circuit-based Boolean reasoning. In *Proceedings of the 38th Design Automation Conference, DAC 2001*, pages 232–237. ACM, 2001.
- [KMS97] H. Kautz, D. McAllester, and B. Selman. Exploiting variable dependency in local search. In *Poster Sessions of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, 1997. Available at <http://www.cs.cornell.edu/home/selman/papers-ftp/97.ijcai.dagsat.ps>.
- [KP80] K. L. Kodandapani and D. K. Pradhan. Undetectability of bridging faults and validity of stuck-at fault test sets. *IEEE Transactions on Computers*, C-29(1):55–59, January 1980.
- [KP94] W. Kunz and D. K. Pradhan. Recursive learning: A new implication technique for efficient solutions to CAD problems—test, verification, and optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(9):1143–1158, September 1994.
- [KPKG02] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai. Robust

- boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(12):1377–1394, December 2002.
- [Kun93] W. Kunz. HANNIBAL: An efficient tool for logic verification based on recursive learning. In *International Conference on Computer Aided Design*, pages 538–543. IEEE Computer Society Press, 1993.
- [Lar92] T. Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design*, 11(1):4–15, January 1992.
- [LWC⁺04] F. Lu, L.-C. Wang, K.-T. T. Cheng, J. Moondanos, and Z. Hanna. A signal correlation guided circuit-SAT solver. *Journal of Universal Computer Science*, 10(12):1629–1654, 2004.
- [LWCH03] F. Lu, L.-C. Wang, K.-T. T. Cheng, and C.-Y. R. Huang. A circuit SAT solver with signal correlation guided learning. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'03)*. IEEE, 2003.
- [Mas98] F. Massacci. Simplification — a general constraint propagation technique for propositional and modal tableaux. In H. de Swart, editor, *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX'98)*, volume 1397 of *Lecture Notes in Artificial Intelligence*, pages 217–231. Springer, 1998.
- [MMZ⁺01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conf.*, pages 530–535, 2001.
- [MR93] N. V. Murray and E. Rosenthal. Dissolution: Making paths vanish. *Journal of the Association for Computing Machinery*, 40(3):504–535, July 1993.
- [MSL07] J. Marques-Silva and I. Lynce. Towards robust CNF encodings of cardinality constraints. In Bessiere [Bes07], pages 483–497.
- [MSS97] J. P. Marques-Silva and K. A. Sakallah. Robust search algorithms for test pattern generation. Technical Report RT/02/97, Dept. of Informatics, Technical University of Lisbon, Lisbon, Portugal, January 1997.
- [MSS99] J. P. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.
- [NP91] T. M. Niermann and J. H. Patel. HITEC: A test generation package for sequential circuits. In *European Conf. on Design Automation*, pages 214–218, 1991.
- [NRW98] A. Nonnengart, G. Rock, and C. Weidenbach. On generating small clause normal forms. In C. Kirchner and H. Kirchner, editors, *Automated Deduction, CADE-15*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 397–411. Springer, 1998.
- [NS96] I. Niemelä and P. Simons. Efficient implementation of the well-founded and stable model semantics. *Fachberichte Informatik 7–96*, Universität Koblenz-Landau, 1996.
- [OS88] F. Oppacher and E. Suen. Harp: A tableau-based theorem prover. *J.*

- Autom. Reasoning*, 4(1):69–100, 1988.
- [Pap95] C. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, 1995.
- [PG86] D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986.
- [PTS07] D. N. Pham, J. Thornton, and A. Sattar. Building structure into local search for SAT. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 2359–2364, 2007.
- [Rot66] J. P. Roth. Diagnosis of automata failures: A calculus and a method. *IBM Journal of Research and Development*, 10(4):278–291, 1966.
- [SB77] T. M. Storey and J. W. Barry. Delay test simulation. In *Design Automation Conf.*, pages 492–494, 1977.
- [SB08] Z. Stachniak and A. Belov. Speeding-up non-clausal local search for propositional satisfiability with clause learning. In *Proceeding of the 11th International Conference on Theory and Applications of Satisfiability Testing (SAT 2008)*, volume 4996 of *Lecture Notes in Computer Science*, pages 257–270. Springer, 2008.
- [SBSV96] P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Trans. on CAD*, 15:1167–1176, 1996.
- [Seb94] R. Sebastiani. Applying GSAT to non-clausal formulas (research note). *J. Artif. Intell. Res.*, 1:309–314, 1994.
- [SFD⁺05] J. Shi, G. Fey, R. Drechsler, A. Glowatz, F. Hapke, and J. Schlöffel. PASSAT: Efficient SAT-based test pattern generation for industrial circuits. In *IEEE Annual Symposium on VLSI*, pages 212–217, 2005.
- [Sin05] C. Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In P. van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, volume 3709 of *Lecture Notes in Computer Science*, pages 827–831. Springer, 2005.
- [Smi85] G. L. Smith. Model for delay faults based upon paths. In *Int'l Test Conf.*, pages 342–349, 1985.
- [SS00] M. Sheeran and G. Stålmarck. A tutorial on stålmarck's proof procedure for propositional logic. *Formal Methods in System Design*, 16(1):23–58, 2000.
- [SSL⁺92] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, University of Berkeley, 1992.
- [Sta02] Z. Stachniak. Going non-clausal. In *5th International Symposium on Theory and Applications of Satisfiability Testing*, 2002.
- [Str00] O. Strichman. Tuning SAT checkers for bounded model checking. In *Computer Aided Verification (CAV 2000)*, volume 1855 of *Lecture Notes in Computer Science*, pages 480–494. Springer, 2000.
- [STS87] M. Schulz, E. Trischler, and T. Sarfert. SOCRATES: A highly efficient automatic test pattern generation system. In *Int'l Test Conf.*, pages 1016–1026, 1987.

- [SVDL04] S. Safarpour, A. Veneris, R. Drechsler, and J. Lee. Managing don't cares in boolean satisfiability. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'04)*, pages 260–265. IEEE, 2004.
- [TBW04] C. Thiffault, F. Bacchus, and T. Walsh. Solving non-clausal formulas with DPLL search. In M. Wallace, editor, *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP 2004)*, volume 3258 of *Lecture Notes in Computer Science*, pages 663–678. Springer, 2004.
- [TG99] P. Tafertshofer and A. Ganz. SAT based ATPG using fast justification and propagation in the implication graph. In *International Conference on Computer Aided Design*, pages 139–146, 1999.
- [TGH97] P. Tafertshofer, A. Ganz, and M. Henftling. A SAT-based implication engine for efficient ATPG, equivalence checking, and optimization of netlists. In *International Conference on Computer Aided Design*, pages 648 – 655, 1997.
- [Vel04] M. N. Velev. Efficient translation of boolean formulas to CNF in formal verification of microprocessors. In *Proceedings of the 2004 Conference on Asia South Pacific Design Automation: Electronic Design and Solution Fair 2004 (ASP-DAC 2004)*, pages 310–315. IEEE, 2004.
- [WA73] M. J. Y. Williams and J. B. Angell. Enhancing testability of large-scale integrated circuits via test points and additional logic. *IEEE Transactions on Computers*, C-22(1):46–60, 1973.
- [WAH03] P. F. Williams, H. R. Andersen, and H. Hulgaard. Satisfiability checking using boolean expression diagrams. *International Journal on Software Tools for Technology Transfer*, 5(1):4–14, November 2003.
- [WLLH07] C.-A. Wu, T.-H. Lin, C.-C. Lee, and C.-Y. R. Huang. QuteSAT: A robust circuit-based SAT solver for complex circuit structure. In *Proceedings of the conference on Design, automation and test in Europe (DATE 2007)*, pages 1313–1318. ACM, 2007.

Chapter 28

Pseudo-Boolean and Cardinality Constraints

Olivier Roussel and Vasco Manquinho

28.1. Introduction

In its broadest sense, a pseudo-Boolean function is a function that maps n Boolean values to a real number. The term pseudo-Boolean comes from the fact that, although these functions are not Boolean, they remain very close to Boolean functions. These pseudo-Boolean functions are studied since the mid 1960s ([HR68]) in the context of Operations Research and 0-1 programming. During the last fifty years, an impressive number of works have been published in this field and this chapter will not even attempt to summarize them (see for example [BH02] for a survey). Instead, this presentation will concentrate on the most recent developments originating from the last advances in the logic community. Specifically, the presentation will be restricted to pseudo-Boolean functions and constraints defined by polynomials with integer coefficients. Imposing that coefficients be integer numbers instead of real numbers is not a concern in practice because applications never require an unlimited precision. Besides, approximate computations on real numbers are not compatible with logical computations.

Pseudo-Boolean functions are a very rich subject of study since numerous problems in various fields (Operations Research, graph theory, combinatorial mathematics, computer science, VLSI design, economics, manufacturing, among others) can be expressed as the problem of optimizing the value of a pseudo-Boolean function. Pseudo-Boolean constraints offer a more expressive and natural way to express constraints than clauses and yet, this formalism remains close enough to the SAT problem to benefit from the recent advances in SAT solving. Simultaneously, pseudo-Boolean solvers benefit from the huge experience in Integer Linear Programming (ILP) and more specifically 0-1 programming. This is particularly true when optimization problems are considered. Powerful inference rules allow to solve some problems polynomially when encoded with pseudo-Boolean constraints while resolution of the problem encoded with clauses requires an exponential number of steps. In a few words, pseudo-Boolean constraints appear as a nice compromise between the expressive power of the formalism used to represent a problem and the difficulty to solve the problem in that formalism.

This chapter will first introduce the definitions of pseudo-Boolean and cardinality constraints and present the decision and optimization problems built on these constraints. The expressive power of the constraints as well as the usual inference rules are then detailed. The next section presents how the most recent SAT algorithms can be transposed to the pseudo-Boolean case and the specific problems posed by this formalism. Specific techniques to solve the optimization problem are also presented. At last, several methods of translating pseudo-Boolean problems to the SAT formalism are studied.

28.2. Basic Definitions

A pseudo-Boolean constraint (PB constraint) is defined over a finite set of Boolean variables x_j . Boolean variables can take only two values *false* and *true* which can be represented equivalently either by $\{\mathbb{F}, \mathbb{T}\}$ or by $\{0, 1\}$. A literal l_j is either a Boolean variable x_j or its negation \bar{x}_j . A positive literal (x_j) evaluates to \mathbb{T} if and only if the corresponding variable x_j is also true, a negative literal (\bar{x}_j) evaluates to \mathbb{T} if and only if the corresponding variable x_j is false.

In pseudo-Boolean constraints, literals are always assigned a constant integer coefficient even if a coefficient of 1 is often omitted in the representation of the constraint. The multiplication of an integer constant by a Boolean variable is defined in a natural way when Booleans are represented by 0 or 1 values. When Boolean are represented by \mathbb{F} or \mathbb{T} values, multiplication is defined equivalently by $\forall a \in \mathbb{Z}, a \cdot \mathbb{T} = a$ and $a \cdot \mathbb{F} = 0$.

28.2.1. Linear Pseudo-Boolean Constraints

A **linear pseudo-Boolean constraint** (LPB constraint) has the following form

$$\sum_j a_j l_j \triangleright b$$

where a_j and b are integer constants, l_j are literals and \triangleright is one of the classical relational operators ($=, >, \geq, <$ or \leq).

The right side of the constraint is often called the **degree** of the constraint.

The addition operator and the relational operators have their usual mathematical meaning. The constraint is satisfied when the left and right term of the constraint evaluate to integers which satisfy the relational operator.

Any constraint which can be rewritten in a straightforward way to match the pattern $\sum_j a_j l_j \triangleright b$ is also considered as a linear pseudo-Boolean constraint.

Example 28.2.1. A first example of a linear pseudo-Boolean constraint is $3x_1 + 4.\bar{x}_2 + 5x_3 \geq 7$. It is satisfied when $x_1 = 1, x_2 = 0$ or $x_1 = 1, x_3 = 1$ or $x_2 = 0, x_3 = 1$.

Here are some other examples:

$$\begin{aligned} x_1 + x_2 + \bar{x}_4 &< 3 \\ 5 + x_1 &= 6 - x_2 \\ 8x_4 + 4x_3 + 2x_2 + x_1 &\leq 8y_4 + 4y_3 + 2y_2 + y_1 \end{aligned}$$

28.2.2. Non-Linear Pseudo-Boolean Constraints

A **non-linear pseudo-Boolean constraint** has the following form

$$\sum_j a_j \cdot \left(\prod_k l_{j,k} \right) \triangleright b$$

where a_j and b are integer constants, $l_{j,k}$ are literals and \triangleright is one of the classical relational operators ($=, >, \geq, <$ or \leq). The semantics of a product of literals is defined in a natural way when Boolean are represented as $\{0, 1\}$ and corresponds equivalently to a logical AND when Booleans are represented as $\{\mathbb{F}, \mathbb{T}\}$. The constraint is satisfied when the left and right term of the constraint evaluate to integers which satisfy the relational operator. Section 28.2.6 gives some methods to translate a non-linear constraint into an equivalent set of linear constraints.

Example 28.2.2. The two pseudo-Boolean constraints below are non-linear:

$$\begin{aligned} 7x_1x_2 + 3x_1 + x_3 + 2x_4 &\geq 8 \\ 7\bar{x}_2 + 3x_1\bar{x}_3 + 2x_4x_2x_1 &\geq 8 \end{aligned}$$

The constraint $x_1 + \bar{x}_1x_2 \geq 1$ is satisfied either when $x_1 = 1$ or when $x_1 = 0$ and $x_2 = 1$. The constraint $x_1 + \bar{x}_1x_2 \geq 2$ is unsatisfiable.

28.2.3. Cardinality Constraints

A **cardinality constraint** is a constraint on the number of literals which are true among a given set of literals. Three cardinality constraints can be defined. The constraint $atleast(k, \{x_1, x_2, \dots, x_n\})$ is true if and only if at least k literals among x_1, x_2, \dots, x_n are true. The constraint $atmost(k, \{x_1, x_2, \dots, x_n\})$ is true if and only if at most k literals among x_1, x_2, \dots, x_n are true. The constraint $exactly(k, \{x_1, x_2, \dots, x_n\})$ is true if and only if exactly k literals among x_1, x_2, \dots, x_n are true. Some obvious relations between these constraints are $atmost(k, \{x_1, x_2, \dots, x_n\}) \equiv atleast(n - k, \{\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n\})$ and $exactly(k, \{x_1, x_2, \dots, x_n\}) \equiv atleast(k, \{x_1, x_2, \dots, x_n\}) \wedge atmost(k, \{x_1, x_2, \dots, x_n\})$. Therefore, the constraint $atleast$ is sufficient to express any cardinality constraint.

The cardinality constraint $atleast(k, \{x_1, x_2, \dots, x_n\})$ translates to the pseudo-Boolean constraint $x_1 + x_2 + \dots + x_n \geq k$ (and the cardinality constraint $atmost(k, \{x_1, x_2, \dots, x_n\})$ translates to $x_1 + x_2 + \dots + x_n \leq k$). Conversely, a pseudo-Boolean constraint where all coefficients are equal such as $\sum_{j=1}^n ax_j \geq b$ is actually a cardinality constraint: $\sum_{j=1}^n ax_j \geq b \equiv atleast(\lceil b/a \rceil, \{x_1, x_2, \dots, x_n\})$ ($\lceil b/a \rceil$ is the smallest integer greater than or equal to b/a). Clearly, a cardinality constraint is a special kind of pseudo-Boolean constraint.

The cardinality constraint $atleast(k, \{x_1, x_2, \dots, x_n\})$ can be translated into an equivalent conjunction of clauses which unfortunately can be of exponential size when no auxiliary variable is introduced [BS94]. The idea of this translation is to express that no more than $n - k$ literals can be false, which means that as soon as $n - k + 1$ literals are selected, at least one of them must be true (this constraint is a clause). So, the constraint $atleast(k, \{x_1, x_2, \dots, x_n\})$ is equivalent to the conjunction of all possible clauses obtained by choosing $n - k + 1$ literals

among $\{x_1, x_2, \dots, x_n\}$. There are obviously $\binom{n}{n-k+1}$ such clauses. The worst case is obtained when $k = n/2 - 1$ and since $\binom{n}{n/2} \geq 2^{n/2}$, this encoding is of exponential size. This means that a cardinality encoding of a problem can be exponentially shorter than a clausal encoding.

Polynomial clausal encodings of cardinality constraints exist [War98, BB03, Sin05] but they all require the introduction of auxiliary variables (see Part 1, Chapter 2).

28.2.4. Clauses

A **clause** $x_1 \vee x_2 \vee \dots \vee x_n$ is by definition equivalent to the cardinality constraint *atleast*(1, $\{x_1, x_2, \dots, x_n\}$) which in turn is equivalent to the pseudo-Boolean constraint $x_1 + x_2 + \dots + x_n \geq 1$. Therefore, clauses are a subset of cardinality constraints which in turn are a subset of all possible pseudo-Boolean constraints. Pseudo-Boolean constraints generalize cardinality constraints which are already a generalization of clauses.

28.2.5. Normalization of Pseudo-Boolean Constraints

Although several relational operators can be used in the definition of a pseudo-Boolean constraint, all pseudo-Boolean constraints can be **normalized** into the following form:

$$\sum_{j=1}^n a_j l_j \geq b, \quad a_j, b \in \mathbb{N}_0^+ \quad (28.1)$$

where l_j denotes either a variable x_j or its complement \bar{x}_j and all coefficients a_j and right-hand side b are non-negative [Bar95a] (a pseudo-Boolean function with only positive coefficients is called *posiform* [BH02]). Moreover, equality constraints or other types of inequality constraints (such as *greater than*, *smaller than* or *smaller than or equal to*), can also be transformed in linear time into *greater than or equal to* constraints.

In order to normalize a pseudo-Boolean constraint, one can start by changing it into a *greater than or equal to* constraint. If the constraint operator is $>$, add 1 to the right-hand side and change it to an inequality with sign \geq . Instead, if the constraint operator is $<$, subtract 1 to the right-hand side and change it to an inequality with sign \leq . Note that these transformations are valid since all coefficients a_j are integers. For equality constraints, one can replace it with two constraints, with the same left and right-hand side, but one with sign \geq and the other with sign \leq . Finally, for constraints with sign \leq , multiply by -1 both the left and right-hand side coefficients and change the sign to \geq .

Next, change the constraint such that all literal coefficients a_j and right-hand-side b are non-negative. For all literals x_j and \bar{x}_j with negative coefficients, replace them with $1 - \bar{x}_j$ and $1 - x_j$, respectively. All literal coefficients are now non-negative and by algebraic manipulation, constant terms can be moved to the right-hand side. If a constraint with a non-positive right-hand side is obtained, that means that the constraint is trivially satisfied.

This procedure allows the mapping in linear time of all pseudo-Boolean constraints into a normalized formulation (28.1). Therefore, to simplify the following

Using only clauses	Using PB constraints
$v \vee \neg l_1 \vee \neg l_2 \vee \dots \vee \neg l_n$	$v \vee \neg l_1 \vee \neg l_2 \vee \dots \vee \neg l_n$
$l_1 \vee \neg v$	$\sum_{k=1}^n l_k - nv \geq 0$
$l_2 \vee \neg v$	
\vdots	
$l_n \vee \neg v$	

Figure 28.1. Two simple methods to linearize an instance by adding extra variables and enforce that $v \Leftrightarrow \prod_{k=1}^n l_k$

descriptions, we will always assume the formulation of pseudo-Boolean constraints in this normalized form.

28.2.6. Linearization Methods

Linearization is a method to convert a non-linear instance into an equivalent linear instance. This is easily done by introducing one extra variable v for each product $\prod_{k=1}^n l_k$. Constraints are added so that v is true if and only if $\prod_{k=1}^n l_k$ is true. A single clause $v \vee \neg l_1 \vee \neg l_2 \vee \dots \vee \neg l_n$ enforces that v be true when all l_k are true. To encode the converse, one can use either n clauses ($\bigwedge_k l_k \vee \neg v$) or use one single pseudo-Boolean constraint $\sum_{k=1}^n l_k - nv \geq 0$. Figure 28.1 summarizes the constraints that are added.

Many linearization methods have been proposed in the context of Operations Research. Fortet's method is probably the oldest one [For60] and is equivalent to the method previously described which introduces one extra variable per product and uses clauses to enforce that the new variable is equivalent to the product. The downside of this kind of method is that a new variable must be introduced for each different product found in the instance. Other methods have been proposed in different contexts to reduce the number of auxiliary variables or even avoid any introduction of auxiliary variables [BM84a, BM84b].

28.3. Decision Problem versus Optimization Problem

Given a formula f which is a conjunction of pseudo-Boolean constraints, one natural question is to identify if this formula is satisfiable or not, that is to say if there exists a valuation of variables such that all constraints are satisfied. This is a decision problem because the only answer which is expected is “yes, it is satisfiable” or “no, these constraints cannot be satisfied”. This **decision problem** is named Pseudo-Boolean Solving (**PBS**). The satisfiability of PB constraints is very close to the SAT problem which deals with the satisfiability of clauses. In such a decision problem, it is not expected in theory that the solver gives a solution when the formula is satisfiable. However, in practice, most solvers which prove satisfiability have identified internally a solution and it is appreciated that they communicate this solution to the user.

Another question which is often dealt with in the pseudo-Boolean context is the optimization problem. Given a formula f which is a conjunction of pseudo-Boolean constraints, decide if this formula has a solution and if it is satisfiable, identify one of the best possible solutions. This is an **optimization problem**: the Pseudo-Boolean Optimization (**PBO**) problem. The solver is expected to answer either “no, these constraints cannot be satisfied” or “yes, the formula is satisfiable and here is the best solution. No other solution can be strictly better”. The best possible solutions are called optimal solutions (or **optimums**) and a problem may have several optimal solutions. In practice, the user is also interested in sub-optimal solutions when a solver does not have enough time to find an optimal solution. When a solver exceeds the given resources, it is usually expected to provide the best solution it has been able to find within its allocated resources. Even if this answer is not the optimum, it is a bound on the optimal solution.

In the optimization problem, there are several ways to identify if a solution is better than another. A common way is to define a function o named objective function which maps each solution S to an integer $o(S)$. Depending on the actual meaning of this function, one may want to minimize its value (in this case, o is called a **cost function** and S is preferred to S' when $o(S) < o(S')$) or one may want to maximize its value (in this case o is called a **utility function**: S is preferred to S' when $o(S) > o(S')$). Since maximizing a function $o(S)$ is equivalent to minimizing $-o(S)$, it is always possible to consider any optimization problem defined by such a function o as a problem of minimizing a cost function.

Therefore, a general representation of the most common pseudo-Boolean optimization problem is:

$$\begin{aligned} & \text{minimize } Cost(x_1, x_2, \dots, x_n) \\ & \text{subject to } \bigwedge_i \sum_j a_{i,j} x_j \geq b_i \end{aligned} \tag{28.2}$$

The cost function is usually defined as a linear function of the pseudo-Boolean variables, that is to say $Cost(x_1, x_2, \dots, x_n) = \sum_j c_j x_j$ with $\forall j, c_j \in \mathbb{Z}$. It may also be defined as a non-linear function of the pseudo-Boolean variables, that is to say $Cost(x_1, x_2, \dots, x_n) = \sum_j c_j \cdot \prod_k x_{j,k}$ with $\forall j, c_j \in \mathbb{Z}$.

The wide use of the optimization problem defined by a linear cost function is undoubtedly linked to the pseudo-Boolean formalism having its roots in the ILP formalism.

Among the many other ways to define that a solution S is better than a solution S' , an interesting approach seems to use several cost functions instead of a single one [LGS⁺07]. Each cost function defines a ranking of solutions based on one criterion and when the criteria cannot be compared, these cost functions cannot be merged as a single cost function. In this context, pareto-optimal solutions are sought. A solution is pareto-optimal if no other solution is at least as good on $n-1$ cost functions and strictly better on the last cost function¹. With this definition, there are now several sets of pareto-optimal solutions which are incomparable.

¹ S is preferred to S' iff $\exists j (\forall i \neq j, Cost_i(S) \leq Cost_i(S') \wedge Cost_j(S) < Cost_j(S'))$

28.3.1. Quadratic Optimization

When the optimization problem is considered, a specific normalization method may be used. Indeed, the optimization of any non-linear instance (with products of any size) can be reduced to the **optimization of a quadratic instance** (with products of size at most 2) by introducing extra variables but no new constraint [BH02].

The idea is to recursively decompose a product of more than two variables by introducing a new variable z which will be forced to be equivalent to the product of two other variables xy . When z is substituted to xy , the size of the product is decreased by one and this can be iterated until the product becomes of size 2.

Instead of introducing new constraints to enforce $z \Leftrightarrow xy$, this approach adds a prevalent penalty to the cost function when $z \not\Leftarrow xy$. Therefore, interpretations where $z \Leftrightarrow xy$ will be preferred to interpretation which do not satisfy this constraint. The penalty function $P(x, y, z) = xy - 2xz - 2yz + 3z$ is such that $P(x, y, z) = 0$ when $z \Leftrightarrow xy$ and $P(x, y, z) > 0$ when $z \not\Leftarrow xy$. For each new variable z , the penalty $c.P(x, y, z)$ is added to the cost function with c being any constant integer greater than the maximum value of the initial cost function. See [HS89, BH02] for an introduction to the huge field of quadratic optimization.

28.4. Expressive Power of Cardinality and Pseudo-Boolean Constraints

28.4.1. Expressiveness

Pseudo-Boolean constraints are more expressive than clauses and a single pseudo-Boolean constraint may replace an exponential number of clauses. More precisely, the translation of a non-trivial pseudo-Boolean constraint to a set of clauses which is semantically equivalent (in the usual sense, i.e. without introducing extra variables) has an exponential complexity in the worst case. One simple example of an exponential case is the constraint $atleast(k, \{x_1, \dots, x_n\})$ which translates to $\binom{n}{n-k+1}$ clauses.

Another simple example of the conciseness of pseudo-Boolean constraints is the encoding of a binary adder. A binary adder is a circuit that sums two integer numbers A and B represented on n bits to obtain a new integer C of $n + 1$ bits. Let a_i, b_i, c_i be the bits of the binary decomposition of respectively A, B, C (such that $A = \sum_{i=0}^{n-1} 2^i a_i$, etc.). The binary adder can be easily encoded as one single pseudo-Boolean constraint: $\sum_{i=0}^{n-1} 2^i a_i + \sum_{i=0}^{n-1} 2^i b_i = \sum_{i=0}^n 2^i c_i$. In contrast, several techniques can be used to encode this problem as clauses but all of them require the introduction of carry variables to obtain an encoding of polynomial size. Several clauses are needed to encode the addition of two bits a_i, b_i and several others are needed to compute the carries. A polynomial encoding of this problem as clauses necessarily uses more variables and more constraints than the pseudo-Boolean encoding. Moreover, it explicitly encodes addition which is otherwise already hard-wired in a pseudo-Boolean solver.

The encoding of the factorization problem is yet another example of the expressive power of PB constraints. Given an integer N , identify two numbers

P and Q such that $N = P.Q$ and $P > 1, Q > 1$ to forbid trivial factorization. The constraint $N = P.Q$ can be encoded as one single non-linear PB constraint: $\sum_i \sum_j 2^{i+j} p_i q_j = N$. Encoding this problem as clauses requires the introduction of extra variables and several binary adders to obtain a polynomial encoding.

28.4.2. Size of Coefficients

As suggested by the factorization example, the coefficients used in a pseudo-Boolean constraint can be of arbitrary size, even in encodings of concrete problems. Since coefficients cannot be assumed to be of bounded size in the general case, the size of the input problem to be considered in complexity analysis must include the size of coefficients. The size of the input formula is $\sum_i (s + size(b_i) + \sum_j (size(a_{i,j}) + size(x_{i,j})))$ where $size(y)$ is the size of the encoding of y , $a_{i,j}$ is the coefficient of the j -th variable in the i -th constraint ($x_{i,j}$) and s is the size of a constraints separator. In contrast, the size of a clausal formula is only $\sum_i (s + \sum_j size(x_{i,j}))$.

A straightforward consequence of this observation is that a solver cannot be correct on any formula if it does not use arbitrary precision arithmetic, which of course has a cost. A solver which uses fixed precision arithmetic will be faced to integer overflows in some cases. These integer overflows may occur during the parsing of the formula. It is easy to check for this case and abort the solver. However, depending on the solver technology, integer overflow may also occur while it infers new constraints from existing constraints. This problem is harder to fix and directly impacts the solver correctness. The use of arbitrary precision arithmetic is the price to pay for correctness on any instance.

28.4.3. Complexity

The satisfiability of pseudo-Boolean constraints (decision problem) is an NP-complete problem. Indeed, checking if an interpretation satisfies a pseudo-Boolean problem can be done in polynomial time with respect to the size of the problem because that size includes the size of coefficients (that check is not polynomial with respect to the sole number of literals in the problem). Therefore, the satisfiability of pseudo-Boolean constraints is in NP. It is also an NP-complete problem since it contains the SAT problem which is NP-complete. The Pseudo-Boolean Optimization problem (PBO) is NP-hard.

Table 28.1 compares the worst case complexity of a few very basic problems with respect to the kind of constraint used (clauses and linear or non-linear pseudo-Boolean constraints using only the *greater than or equal to* operator). The difference of complexity illustrates both the expressive power of pseudo-Boolean constraints and the difficulty to obtain efficient methods to solve pseudo-Boolean problems.

Deciding if a single clause is satisfiable is a trivial problem. The only check to do is to verify that it is not empty (constant time). Checking if a *greater than or equal to* LPB constraint is satisfiable is slightly more complex since it requires to compute the bounds of the values of the left side which has a linear complexity.

Table 28.1. Comparison of the worst case complexity of a few problems with respect to the kind of constraint used

	Clauses	Linear PB constraints (\geq)	Non-linear PB constraints (\geq)
satisfiability of 1 constraint	$O(1)$	$O(n)$	NP-complete
satisfiability of 2 constraints	$O(n)$	NP-complete	NP-complete

Deciding the satisfiability of one single *greater than or equal to* non-linear constraint is much harder since it is a NP-complete problem. Indeed, any 3-SAT instance can be linearly transformed into a single *greater than or equal to* constraint with non-linear terms. Each clause $C = l_1 \vee l_2 \vee l_3$ can be transformed into a constraint $f(C) = 1$ where $f(C) = l_1 + l_2 + l_3 - l_1l_2 - l_1l_3 - l_2l_3 + l_1l_2l_3$. The constraint $f(C) = 1$ is true if and only if the clause is satisfied (otherwise, $f(C) = 0$). The whole 3-SAT formula $\bigwedge_i C_i$ is transformed into the equivalent non-linear pseudo-Boolean constraint $\sum_i f(C_i) \geq n$ where n is the number of clauses in the formula.

Deciding if two clauses are satisfiable amounts to finding a literal in each clause such that the chosen literals are not complementary. This can be done in linear time.

The satisfiability of two pseudo-Boolean constraints (linear or non-linear) is a NP-complete problem because the subset sum problem can be encoded as two *greater than or equal to* PB constraints. The subset sum problem consists in determining if a subset of a given set of integer can be summed to obtain a given constant integer. This problem is known to be NP-complete. Let $\{a_0, a_1, \dots, a_n\}$ and k be the input of the subset sum problem. This problem is trivially encoded by the linear pseudo-Boolean constraint $\sum_{i \in [0..n]} a_i x_i = k$ where x_i indicates whether a_i should be in the chosen subset or not. This equality constraint can be rewritten as two *greater than or equal to* linear constraints.

28.5. Inference Rules

This section presents a review on the main inference rules that can be used on pseudo-Boolean constraints. It is assumed that constraints have been normalized to use strictly positive coefficients (generalization to non-normalized constraints is straightforward).

We first review some fundamental rules which are either definitions or immediately derived from the axioms.

$$\frac{}{\bar{x} = 1 - x} \quad (\text{negation})$$

$$\frac{}{x.x = x} \quad (\text{idempotence})$$

$$\frac{}{x \geq 0} \quad (bounds)$$

$$\frac{}{-x \geq -1}$$

The negation rule allows the normalization of constraints so that they contain only positive coefficients. The idempotence rule implies that no exponent is needed in a product of literals. At last, the bounds rule is a direct implication from the axiom $x = 0 \vee x = 1$ and is used to obtain a few simplification rules presented below.

The following rules can be applied to both linear and non-linear constraints. Therefore, L_j will denote either a literal or a product of literals. By definition, L_j is either 0 or 1 and therefore $0 \leq L_j \leq 1$.

The next three rules are the ones used in the **cutting plane proof system**.

$$\frac{\sum_j a_j L_j \geq b \quad \sum_j c_j L_j \geq d}{\sum_j (a_j + c_j) L_j \geq b + d} \quad (\text{addition})$$

$$\frac{\sum_j a_j L_j \geq b \quad \alpha > 0 \quad \alpha \in \mathbb{N}}{\sum_j \alpha a_j L_j \geq \alpha b} \quad (\text{multiplication})$$

$$\frac{\sum_j a_j L_j \geq b \quad \alpha > 0}{\sum_j \lceil \frac{a_j}{\alpha} \rceil L_j \geq \lceil \frac{b}{\alpha} \rceil} \quad (\text{division})$$

These three rules form a complete proof system[Gom63], which means that whenever the constraints are unsatisfiable, these rules allow to derive the contradiction $0 \geq 1$.

It is important to notice that, even when all initial constraints are cardinality constraints, the addition rule cannot be restricted to generate only cardinality constraints. In other words, proof of unsatisfiability of a set of cardinality constraints requires in general the use of generic intermediate pseudo-Boolean constraints. As an example, the conjunction of the two cardinality constraints $\{a + b + c + d \geq 3, a + \bar{b} + \bar{c} + \bar{d} \geq 3\}$ is unsatisfiable. This cannot be proved without inferring by the addition rule $2a \geq 3$ which is not a cardinality constraint.

The multiplication rule is essentially a commodity rule since it can be replaced by iterated applications of the addition rule. The condition $\alpha \in \mathbb{N}$ in the multiplication rule is only used to obtain a constraint which still has integer coefficients. The multiplication and division rules could be unified as one single rule where α is not required to be an integer.

$$\frac{\sum_j a_j L_j \geq b \quad \alpha > 0}{\sum_j \lceil \alpha a_j \rceil L_j \geq \lceil \alpha b \rceil} \quad (\text{multiplication/division})$$

$\lceil r \rceil$ is the ceiling function which returns the smallest integer number which is greater than or equal to r . The correctness of this rule derives from $\forall r_j \in \mathbb{R}, \sum_j \lceil r_j \rceil \geq \lceil \sum_j r_j \rceil$.

The addition and multiplication rules are often combined together in a single rule which allows to infer a new constraint with one or more variables eliminated.

$$\frac{\begin{array}{l} \sum_j a_j L_j \geq b \\ \sum_j c_j L_j \geq d \\ \alpha \in \mathbb{N}, \beta \in \mathbb{N} \\ \alpha > 0, \beta > 0 \end{array}}{\sum_j (\alpha a_j + \beta c_j) L_j \geq \alpha b + \beta d} \quad (\text{generalized resolution})$$

Usually, α and β are chosen in order to eliminate at least one of the literals from the inferred rule. If $p.l$ appears in the first constraint and $q.\bar{l}$ in the second, choosing $\alpha = q/\text{GCD}(p, q)$ and $\beta = p/\text{GCD}(p, q)$ will generate in the sum the term $p.q/\text{GCD}(p, q).(l+\bar{l})$ which is a constant since $l+\bar{l} = 1$. It should be noticed that several literals can be removed simultaneously. For example, eliminating x_1 in $\{x_1 + x_2 + x_3 \geq 1, 2\bar{x}_1 + 2\bar{x}_2 + x_4 \geq 3\}$ also eliminates x_2 and generates $2x_3 + x_4 \geq 1$ which will be simplified to $x_3 + x_4 \geq 1$ by the saturation rule.

The saturation rule indicates that when the constraint is normalized and one of the coefficients is larger than the degree (the right side of the inequation), it can be truncated to become equal to the degree.

$$\frac{\begin{array}{l} \sum_j a_j L_j \geq b \\ \forall j, a_j \geq 0 \\ a_k > b \end{array}}{bL_k + \sum_{j \neq k} a_j L_j \geq b} \quad (\text{saturation})$$

From a logical point of view, this rule is straightforward. When $a_j > b$, assigning true to L_j is enough to satisfy the constraint and indeed, any coefficient greater than or equal to b will have the same effect and the saturation rule simply chooses the least of these coefficients. This saturation rule can be replaced by iterated applications of the multiplication/division rule with α chosen so that any coefficient strictly greater than b will be reduced at least by 1 and any coefficient smaller or equal to b will not be modified (these two conditions are satisfied when $(b-1)/b < \alpha \leq b/(b+1)$).

It must be noticed that the generalized resolution rule is not sufficient to obtain a complete proof system because it cannot remove all non-integer solutions. For example, generalized resolution will not infer inconsistency from $\{a \vee b, a \vee \neg b, \neg a \vee b, \neg a \vee \neg b\}$ because $a = 1/2, b = 1/2$ is a solution of the initial constraints and all constraints generated by generalized resolution. The saturation rule (or equivalently the division rule) is required to cut the non-integer solutions.

The last rules are just a direct application of the addition rule and the bounds axioms but appear particularly useful in some constraint driven algorithms. These rules (weakening rules) allow to remove some occurrences of a literal in a constraint. However, the derived constraint is in general weaker than the initial

constraint (i.e. it forbids fewer interpretations than the initial constraint). The derived constraint may even become a tautology.

$$\frac{\sum_j a_j L_j \geq b}{a_k > 0} \quad (\text{weakening})$$

$$\frac{\sum_{j \neq k} a_j L_j \geq b - a_j}{\sum_j a_j L_j \geq b}$$

$$\frac{\sum_j a_j L_j \geq b}{a_k > 0}$$

$$\frac{a_k > a > 0}{(a_k - a)L_j + \sum_{j \neq k} a_j L_j \geq b - a} \quad (\text{partial weakening})$$

Since each inference rule which was presented can be derived from the rules of the cutting plane proof system (addition, multiplication, division) and the axioms, the term “cutting plane rules” will be used indistinctly in the rest of this chapter to refer to the application of one or several of these rules.

28.5.1. Inference Power

28.5.1.1. Comparison with SAT Inference Rules

Two inference rules are used in propositional calculus: resolution and merging.

$$\frac{l \vee A \quad \neg l \vee B}{\neg \exists l' \text{ s.t. } l' \in A \wedge \neg l' \in B} \quad (\text{resolution})$$

$$\frac{A \vee B}{A \vee B}$$

$$\frac{l \vee l \vee C}{l \vee C} \quad (\text{merging})$$

If the resolution rule is well known, the merging rule is often ignored because usually, clauses are defined as sets of literals and the merging rule is directly obtained from the definition of a set. This rule is more visible in predicate calculus where a most general unifier must be computed to merge literals. However, merging is actually required to obtain a complete proof system in propositional calculus.

The resolution rule in propositional calculus directly corresponds to the addition rule. The two clauses $l \vee A$ and $\neg l \vee B$ correspond to the PB constraints $l + \sum_j l_j^A \geq 1$ and $1 - l + \sum_j l_j^B \geq 1$. The sum of these constraints is $\sum_j l_j^A + \sum_j l_j^B \geq 1$. Provided A and B does not contain a pair of complementary literals, this sum corresponds to a clause (possibly with some literals repeated if some l_j^A is equal to a l_j^B). If A and B contained a pair of complementary literals, these literals would disappear and the degree of the PB constraints would become 0 which corresponds to a tautology.

The merging rule is a straightforward application of the saturation rule. $l \vee l \vee C$ corresponds to $2l + \sum_j l_j^C \geq 1$ which is reduced by the saturation rule as $l + \sum_j l_j^C \geq 1$ which represents $l \vee C$.

This comparison proves that the cutting plane proof system can polynomially simulate the resolution proof system. This means that the cutting plane proof system is at least as powerful as the resolution proof system.

28.5.1.2. The Cutting Plane System is more Powerful than Resolution: The Pigeon-Hole Example

This section presents an example of a problem which has a polynomial proof in the cutting plane proof system while no polynomial proof exists in the resolution proof system. This proves that the cutting plane proof system is strictly more powerful than the resolution proof system. This example problem is the well-known pigeon-hole problem. Interestingly, this example will show that conflict-driven constraint learning solvers (described in Part 1, Chapter 4) can easily discover this polynomial proof.

The pigeon-hole problem consists in placing p pigeons in h holes such that each hole contains at most one pigeon and each pigeon is placed in at most one hole. This second constraint can be omitted when only the satisfiability of the problem is considered. This problem is well known since it was used to prove the intractability of resolution [Hak85]. In its clausal form, an exponential number of resolution steps is required to prove the unsatisfiability of the problem when $p > h$.

However, when cardinality constraints and the cutting plane proof system are used, a polynomial number of inferences suffices to prove unsatisfiability and a conflict-driven constraint learning solver can easily find this proof.

As an example, let us consider the pigeon-hole problem with $p = 5$ and $h = 4$ (5 pigeons, only 4 holes) in its satisfiability oriented encoding. Let $P_{p,h}$ be a Boolean variable which is true if and only if pigeon number p is placed in hole number h . This problem can be encoded by the following cardinality constraints:

$$P_{1,1} + P_{1,2} + P_{1,3} + P_{1,4} \geq 1 \quad (\text{P1})$$

$$P_{2,1} + P_{2,2} + P_{2,3} + P_{2,4} \geq 1 \quad (\text{P2})$$

$$P_{3,1} + P_{3,2} + P_{3,3} + P_{3,4} \geq 1 \quad (\text{P3})$$

$$P_{4,1} + P_{4,2} + P_{4,3} + P_{4,4} \geq 1 \quad (\text{P4})$$

$$P_{5,1} + P_{5,2} + P_{5,3} + P_{5,4} \geq 1 \quad (\text{P5})$$

$$\overline{P_{1,1}} + \overline{P_{2,1}} + \overline{P_{3,1}} + \overline{P_{4,1}} + \overline{P_{5,1}} \geq 4 \quad (\text{H1})$$

$$\overline{P_{1,2}} + \overline{P_{2,2}} + \overline{P_{3,2}} + \overline{P_{4,2}} + \overline{P_{5,2}} \geq 4 \quad (\text{H2})$$

$$\overline{P_{1,3}} + \overline{P_{2,3}} + \overline{P_{3,3}} + \overline{P_{4,3}} + \overline{P_{5,3}} \geq 4 \quad (\text{H3})$$

$$\overline{P_{1,4}} + \overline{P_{2,4}} + \overline{P_{3,4}} + \overline{P_{4,4}} + \overline{P_{5,4}} \geq 4 \quad (\text{H4})$$

The first five constraints (P1 to P5) encode that a pigeon must be in at least one hole and the last four constraints (H1 to H4) encode that a hole can contain at most one pigeon (H1 is a normalized form of $P_{1,1} + P_{2,1} + P_{3,1} + P_{4,1} + P_{5,1} \leq 1$).

A conflict-driven constraint learning solver may start by assigning true to $P_{1,1}$ at level 1 of the decision tree ($v@l$ will denote that the solver assigns value v to a variable at the decision level l , so $P_{1,1}$ is assigned $1@1$). By H1, $P_{2,1}$, $P_{3,1}$, $P_{4,1}$, $P_{5,1}$ are all set to false. Since (P1) is now satisfied, the next decision variable can be $P_{2,2}$ at level 2. Once this variable is assigned true, H2 implies that $P_{1,2}$, $P_{3,2}$, $P_{4,2}$, $P_{5,2}$ are all false. Next, $P_{3,3}$ may be assigned true at level 3. This implies that $P_{1,3}$, $P_{2,3}$, $P_{4,3}$, $P_{5,3}$ are all false (by H3), $P_{4,4}$ is true (by P4) and then $P_{1,4}$,

Literal	val@lvl	reason	Literal	val@lvl	reason	Literal	val@lvl	reason	Literal	val@lvl	reason
$P_{1,1}$	1@1	—	$P_{1,1}$	1@1	—	$P_{1,1}$	1@1	—	$P_{1,1}$	0@0	L3
$P_{2,1}$	0@1	H1	$P_{2,1}$	0@1	H1	$P_{2,1}$	0@1	H1	$P_{1,2}$	0@0	L3
$P_{3,1}$	0@1	H1	$P_{3,1}$	0@1	H1	$P_{3,1}$	0@1	H1	$P_{1,3}$	0@0	L3
$P_{4,1}$	0@1	H1	$P_{4,1}$	0@1	H1	$P_{4,1}$	0@1	H1	$P_{1,4}$	0@0	L3
$P_{5,1}$	0@1	H1	$P_{5,1}$	0@1	H1	$P_{5,1}$	0@1	H1			
$P_{2,2}$	1@2	—	$P_{2,2}$	1@2	—	$P_{1,2}$	0@1	L2			
$P_{1,2}$	0@2	H2	$P_{1,2}$	0@2	H2	$P_{1,3}$	0@1	L2			
$P_{3,2}$	0@2	H2	$P_{3,2}$	0@2	H2	$P_{1,4}$	0@1	L2			
$P_{4,2}$	0@2	H2	$P_{4,2}$	0@2	H2	$P_{2,2}$	0@1	L2			
$P_{5,2}$	0@2	H2	$P_{5,2}$	0@2	H2	$P_{2,3}$	0@1	L2			
$P_{3,3}$	1@3	—	$P_{1,3}$	0@2	L1	$P_{2,4}$	0@1	L2			
$P_{1,3}$	0@3	H3	$P_{1,4}$	0@2	L1						
$P_{2,3}$	0@3	H3	$P_{2,3}$	0@2	L1						
$P_{4,3}$	0@3	H3	$P_{2,4}$	0@2	L1						
$P_{5,3}$	0@3	H3	$P_{3,3}$	0@2	L1						
$P_{4,4}$	1@3	P4	$P_{3,4}$	0@2	L1						
$P_{1,4}$	0@3	H4									
$P_{2,4}$	0@3	H4									
$P_{3,4}$	0@3	H4									
$P_{5,4}$	0@3	H4									

(a) first conflict

(b) second conflict

(c) third conflict

(d) final step

Figure 28.2. Summary of the decisions and propagations in the pigeon-hole example. Each line in a table represents which value (val) is assigned to which literal at which level (lvl) and what is the reason of this assignment.

$P_{2,4}$, $P_{3,4}$, $P_{5,4}$ are false (by H4). Figure 28.2(a) summarizes the decisions and propagations which occurred. The reason of each propagation is indicated in that figure as well as the decision level of each propagation or decision.

At this point, constraint P5 is falsified and conflict analysis can start. The first step is to combine the conflicting constraint (P5) $\overline{P_{5,1}} + \overline{P_{5,2}} + \overline{P_{5,3}} + \overline{P_{5,4}} \geq 1$ with the reason of the last propagated literal (H4) $\overline{P_{1,4}} + \overline{P_{2,4}} + \overline{P_{3,4}} + \overline{P_{4,4}} + \overline{P_{5,4}} \geq 4$ to eliminate the propagated variable. This yields a new constraint $\overline{P_{5,1}} + \overline{P_{5,2}} + \overline{P_{5,3}} + \overline{P_{1,4}} + \overline{P_{2,4}} + \overline{P_{3,4}} + \overline{P_{4,4}} \geq 4$. Since this constraint will not propagate any literal when we backtrack to level 2, conflict analysis goes on. Another resolution step is performed with (P4) which is the reason of the propagation of $P_{4,4}$ to obtain $\overline{P_{4,1}} + \overline{P_{4,2}} + \overline{P_{4,3}} + \overline{P_{5,1}} + \overline{P_{5,2}} + \overline{P_{5,3}} + \overline{P_{1,4}} + \overline{P_{2,4}} + \overline{P_{3,4}} \geq 4$. Once again, this constraint implies no literal at level 2. Therefore, it is combined with (H3) to eliminate $\overline{P_{4,3}}$ and $\overline{P_{5,3}}$. This yields constraint (L1) $\overline{P_{4,1}} + \overline{P_{4,2}} + \overline{P_{5,1}} + \overline{P_{5,2}} + \overline{P_{1,3}} + \overline{P_{1,4}} + \overline{P_{2,3}} + \overline{P_{2,4}} + \overline{P_{3,3}} + \overline{P_{3,4}} \geq 6$ which will propagate literals when the solver backtracks to level 2. Conflict analysis stops and this constraint is learnt.

Figure 28.3 represents the derivation of constraint L1 from the conflict constraint and the different reasons of propagations. Literals that are erased in a resolution step are written in parentheses.

The first inferred constraint in this derivation presented in figure 28.3 is a nice illustration of the difference between the SAT encoding and the cardinality encoding of the pigeon-hole problem. In a SAT encoding, the reason of the propagation of $\overline{P_{5,4}}$ would be the clause $\neg P_{4,4} \vee \neg P_{5,4}$ and the resolvent with the conflict would be $\overline{P_{5,1}} \vee \overline{P_{5,2}} \vee \overline{P_{5,3}} \vee \neg P_{4,4}$. The semantics of this clause can be expressed as 'when $P_{4,4}$ is true, one of $P_{5,1}$, $P_{5,2}$ or $P_{5,3}$ must be true'. In contrast, the semantics of the pseudo-Boolean constraint (in combination with H4)

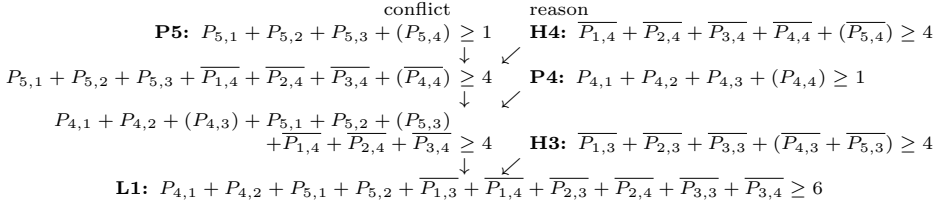


Figure 28.3. First conflict analysis in the pigeon-hole example

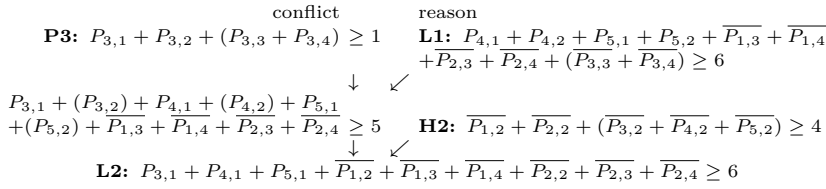


Figure 28.4. Second conflict analysis in the pigeon-hole example

is 'when any of $P_{1,4}, P_{2,4}, P_{3,4}, P_{4,4}$ is true, then one of $P_{5,1}, P_{5,2}$ or $P_{5,3}$ must be true'. The strength of the pseudo-Boolean formulation in this case is that one PB constraint replaces several clauses and this single constraint captures several symmetries of the problem.

After the first conflict analysis, the solver backtracks to level 2. At this level, $P_{4,1}, P_{4,2}, P_{5,1}, P_{5,2}$ are all false. Therefore, constraint (L1) propagates $P_{1,3}, P_{1,4}, P_{2,3}, P_{2,4}, P_{3,3}, P_{3,4}$ equal false but since $P_{3,1}, P_{3,2}$ are already false, (P3) $P_{3,1} + P_{3,2} + P_{3,3} + P_{3,4} \geq 1$ is a new conflict. Figure 28.2(b) presents the propagations just before the conflict and figure 28.4 represents the conflicts analysis which leads to constraint (L2) which is learnt.

When the solver backtracks to level 1, $P_{3,1}, P_{4,1}, P_{5,1}$ are false and therefore constraint (L2) implies that $P_{1,2}, P_{1,3}, P_{1,4}, P_{2,2}, P_{2,3}, P_{2,4}$ are false. But since $P_{2,1}$ is false, (P2) $P_{2,1} + P_{2,2} + P_{2,3} + P_{2,4} \geq 1$ is a new conflict. Figure 28.2(c) shows the assignment stack just before the conflict is detected and figure 28.5 presents the resulting conflicts analysis.

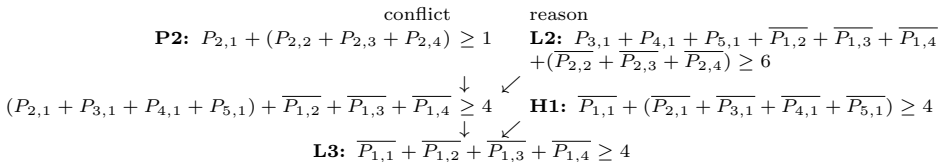


Figure 28.5. Third conflict analysis in the pigeon-hole example

Algorithm 28.1 Generic Pseudo-Boolean Solving Algorithm

```

1: function GENERIC_PB_SOLVER(P)
2:   while true do
3:     if Solution_Found(P) then
4:       return SATISFIABLE
5:     else
6:       Decide(P)
7:     end if
8:     while Deduce(P)=CONFLICT do
9:       if Diagnose(P)=CONFLICT then
10:        return UNSATISFIABLE
11:      end if
12:    end while
13:   end while
14: end function

```

When the solver backtracks to level 0, constraint (L3) implies that $P_{1,1}$, $P_{1,2}$, $P_{1,3}$, $P_{1,4}$ are all false, but this assignment conflicts with $P_{1,1} + P_{1,2} + P_{1,3} + P_{1,4} \geq 1$. Figure 28.2(d) presents the assignments when the solver backtracks to level 0. Since there is no decision that can be changed at level 0, the solver has proved unsatisfiability, and this required only a polynomial number of steps.

28.6. Current Algorithms

Due to the recent developments in algorithms for Propositional Satisfiability (SAT), algorithms for Pseudo-Boolean Solving (PBS) and Optimization (PBO) have also been the subject of thorough research work, mainly in applying generalizations of successful ideas already used in SAT algorithms. Nevertheless, other techniques used in Operations Research algorithms have also been proposed.

In this section, the main algorithmic techniques for solving pseudo-Boolean formulations are described. We start by presenting the generalization of commonly used techniques in SAT solvers, namely Boolean constraint propagation and conflict-driven learning and backtracking. Finally, we focus on specific techniques for pseudo-Boolean optimization, with emphasis on lower bounding procedures and cutting plane generation.

Algorithm 28.1 presents the pseudo-code for a generic pseudo-Boolean solver based on a backtrack search approach where a depth-first tree is built to explore all possible assignments to the problem variables. Each branch of the search tree corresponds to a partial assignment to problem variables.

The algorithm starts with an empty search tree (no assigned variables) and at each step the **Decision** procedure extends a branch of the search tree (partial assignment) by assigning one of the unassigned variables from problem instance P . Next, the **Deduce** procedure identifies necessary assignments by applying Boolean constraint propagation [DP60, ZM88]. This procedure can also be modified in order to apply problem reduction techniques [Cou93, VKBSV97]. If there exists a conflict, i.e. an unsatisfied constraint, the **Deduce** procedure returns **CONFLICT**. Whenever a conflict occurs, **Diagnose** applies the conflict analysis procedure to resolve the conflict and backtrack in the search tree. When the algorithm is unable to backtrack any further, the search stops and returns **UNSATISFIABLE**.

If at any step of the search, all constraints from P are satisfied, the algorithm returns SATISFIABLE.

28.6.1. Boolean Constraint Propagation

The **unit clause rule** [DP60] applied to propositional clauses in SAT algorithms, states that given a partial assignment and a unit clause ω (with one unassigned literal l_j and all other literals assigned value 0), the unassigned literal l_j in the unit clause must have value 1. Hence, the associated variable assignment can be added to the current partial assignment. The successive application of the unit clause rule is known as **Boolean constraint propagation** or **unit propagation** [DP60, ZM88]. A generalized version of Boolean constraint propagation can also be applied to pseudo-Boolean constraints, using a similar definition of unit constraint. The main difference is that more than one literal might be assigned value 1 due to a single pseudo-Boolean constraint.

Consider a pseudo-Boolean constraint ω in normalized form. Given a partial assignment, let ω^v denote the set of literals in constraint ω with value $v \in \{0, 1\}$. Let also L_ω be the current value on the left hand side of the constraint defined as:

$$L_\omega = \sum_{l_j \in \omega^1} a_j \quad (28.3)$$

Finally, let s_ω denote the slack of constraint ω defined as:

$$s_\omega = \sum_{l_j \notin \omega^0} a_j - b \quad (28.4)$$

The slack represents the maximal amount by which the left side may exceed the right side of the constraint (when all yet unassigned variables are set to true). Given a partial assignment, a constraint ω is said to be *satisfied* if $L_\omega \geq b$. In contrast, if $s_\omega < 0$, ω is said to be *unsatisfied*. In this case, it is guaranteed that with the current partial assignment, ω cannot be satisfied. A constraint ω is said to be *unresolved* if $L_\omega < b$ and $s_\omega \geq 0$. This means that ω can still become satisfied or unsatisfied, depending on the assignments to currently unassigned variables. A special case of the unresolved constraints is when there is at least one unassigned literal $l_j \in \omega$ such that $s_\omega - a_j < 0$. Then ω is said to be a *unit* constraint since literal l_j *must* be assigned value 1 in order for ω to become satisfied. Observe that in a unit pseudo-Boolean constraint ω , more than one literal might have to be assigned value 1 for the constraint to be satisfied. It is possible that after assigning value 1 to l_j , pseudo-Boolean constraint ω remains unit and other literals must be assigned value 1.

Example 28.6.1. Let ω be the constraint $5x_1 + 3x_2 + 3x_3 + x_4 \geq 6$. When all variables are unassigned, $L_\omega = 0$ and $s_\omega = 6$. This constraint is unresolved (i.e. not yet satisfied but some affectations of the variables may still satisfy the constraint because $s_\omega \geq 0$). If x_1 is assigned false, $L_\omega = 0$ and $s_\omega = 1$. The constraint is unresolved but since $s_\omega - a_2 < 0$, x_2 must be assigned true otherwise the constraint becomes unsatisfied. Once x_2 is assigned true, $L_\omega = 3$ and $s_\omega = 1$. The constraint is still unresolved and since $s_\omega - a_3 < 0$, x_3 must also

be assigned true. Now, $L_\omega = 6$ and $s_\omega = 1$, the constraint is now satisfied and propagation stops. In this example, the assignment of one variable (x_1) implied the propagation of two other variables (x_2 and x_3) which in this particular case is enough to satisfy the constraint. This example also illustrates that some variables (x_4) can remain unassigned after the propagation, which does not happen with clauses.

A solver can incrementally compute the slack of each constraint to detect unit constraints. This is the analog of the counter scheme formerly used in SAT. But in order to engineer a good pseudo-Boolean solver, it is necessary to quickly detect when a constraint becomes unit for Boolean constraint propagation or unsatisfied to backtrack in the search tree. Therefore, the use of efficient data structures to manipulate problem constraints is vital. Propositional clauses are a particular case of pseudo-Boolean constraints and specific data structures have been developed for SAT solvers in order to efficiently manipulate these constraints. More recently, new data structures have also been proposed for other types of pseudo-Boolean constraints.

The purpose of **lazy data structures** is to be able to easily detect when a constraint becomes unsatisfied (to backtrack in the search tree) or unit (to detect necessary assignments) while avoiding unnecessary processing on unresolved constraints. As described in Part 1, Chapter 4, in propositional clauses, watching two literals is enough to accomplish these objectives and several strategies for choosing which literals to watch have already been developed [Zha97, MMZ⁺01, LMS05]. However, in cardinality constraints and general pseudo-Boolean constraints, a larger number of literals must be watched in order to be able to detect when a constraint becomes unsatisfied or unit.

Let W denote the set of **watched literals** in a pseudo-Boolean constraint ω and let $S(W)$ denote the sum of the watched literal coefficients:

$$S(W) = \sum_{l_j \in W} a_j \quad (28.5)$$

In order to detect that constraint ω becomes unsatisfied, it is only necessary to watch a set of literals W (unassigned or assigned value 1) such that $S(W) \geq b$, since it is still possible to satisfy ω by assigning value 1 to the unassigned watched literals. However, to make sure that a constraint does not imply a necessary assignment, one must watch a set of literals W such that:

$$S(W) \geq b + a_{max} \quad (28.6)$$

where a_{max} is greater than or equal to the largest coefficient of unassigned literals in the constraint. Note that if (28.6) is true, then there is no unassigned literal l_j such that $S(W) - a_j < b$. Hence, the constraint is not unit.

Whenever a watched literal $l_j \in W$ is assigned value 0, l_j is removed from W and unassigned or assigned value 1 literals are added to W until (28.6) holds. In this case, the constraint remains unresolved or becomes satisfied. However, if all non-value 0 literals are being watched and $S(W) < b$, then the constraint is unsatisfied. Otherwise, if $S(W) < b + a_{max}$ then the constraint is unit and the necessary assignments can be implied until (28.6) is true.

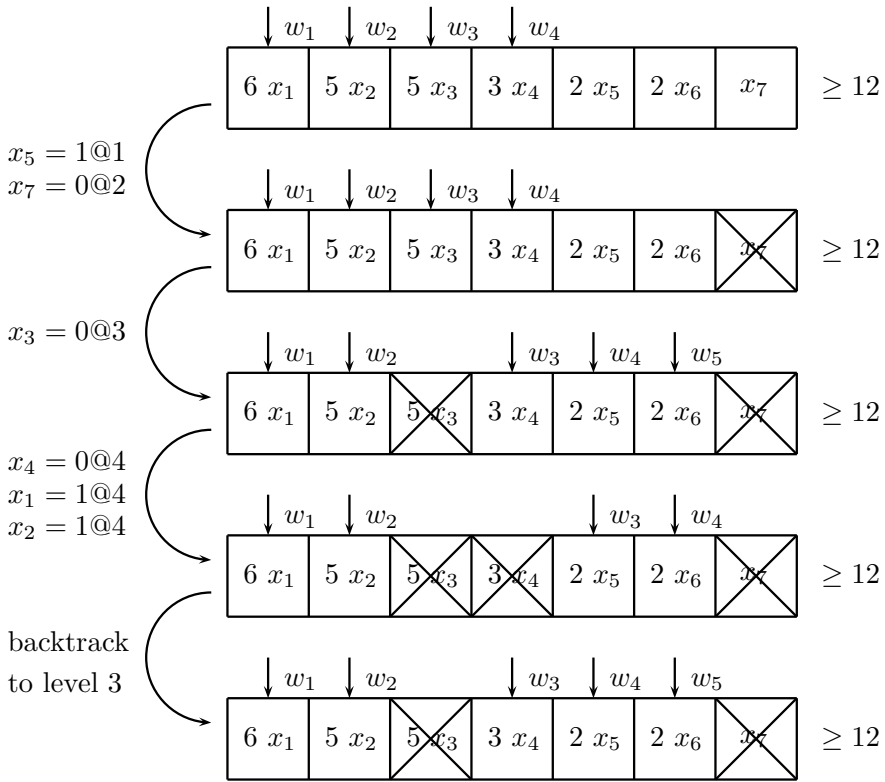


Figure 28.6. Example of lazy data structure for pseudo-Boolean constraints

Figure 28.6 presents an example of the use of lazy data structures in general pseudo-Boolean constraints. Consider the following pseudo-Boolean constraint:

$$6x_1 + 5x_2 + 5x_3 + 3x_4 + 2x_5 + 2x_6 + x_7 \geq 12 \tag{28.7}$$

and suppose that initially we have $W = \{x_1, x_2, x_3, x_4\}$. Note that watching only 3 literals would not satisfy (28.6) and consequently would not be enough to detect all situations of implication of necessary assignments.

As in the watched literals approach for propositional clauses, if non-watched literals are assigned, no update is necessary in the data structure. This is illustrated in Figure 28.6 if we start by assigning x_5 and x_7 . Afterwards, when x_3 is assigned value 0, the set of watched literals must be updated and we have $W = \{x_1, x_2, x_4, x_5, x_6\}$ such that (28.6) is satisfied. Next, if x_4 is assigned value 0, we can deduce value 1 to x_1 and x_2 . Afterwards, when search backtracks to level 3, no update to the watched literals is necessary.

One should note that for cardinality constraints, the number of watched literals in this approach is constant and the size of W is always $b + 1$. As expected,

for propositional clauses the size of W is always 2. However, as shown in our example, for general pseudo-Boolean constraints, the number of watched literals can vary since the value of a_{max} depends on which literals are assigned at any given moment.

The major drawback of this approach for handling general pseudo-Boolean constraints is the fact that most of the time is lost in updating the value of a_{max} [SS05]. Other options have already been proposed regarding the value of a_{max} to be used, as for example fixing the value of a_{max} to the value of the highest literal coefficient in the constraint [SS05]. With this approach, no update on the value of a_{max} is necessary. However, during the search process, more literals than necessary might be watched.

28.6.2. Conflict Analysis

Besides Boolean constraint propagation, other techniques from state of the art SAT algorithms can also be used in pseudo-Boolean algorithms. One of the most important is the **conflict analysis** procedure to be applied during search whenever a constraint becomes unsatisfied by the current partial assignment under consideration. As with SAT solving, it is possible to do non-chronological backtracking in a backtrack search pseudo-Boolean solver by using a sophisticated conflict analysis procedure.

The advantage of using non-chronological backtracking is that this technique might prevent the algorithm to waste a significant amount of time exploring a useless region of the search space only to discover that the region does not contain any satisfying assignment. Algorithms that perform non-chronological backtracks can jump directly from the current decision level d to another preceding decision level d' where $d - d' > 1$. The decision level d' where it is safe to backtrack to is determined by a conflict analysis procedure.

One approach to the conflict analysis procedure is to apply the same ideas as in SAT algorithms, in which an analysis of an **implication graph** represented as an directed acyclic graph (DAG) is made. The implication graph is defined as follows ($v(x)$ represents the value of variable x and $\delta(x)$ represents the decision level where x was assigned, notation $v(x_j)@ \delta(x_j)$ is used to indicate the value of the variable and the level at which it was assigned):

- Each vertex corresponds to a variable assignment $x_j = v(x_j)@ \delta(x_j)$
- The predecessors of a vertex $x_j = v(x_j)@ \delta(x_j)$ in the graph are the associated assignments to the literals assigned value 0 in the unit constraint ω that led to the implication of value $v(x_j)$ to x_j . The directed edges from the predecessors vertices to vertex $x_j = v(x_j)@ \delta(x_j)$ are all labeled with ω . Vertices that have no predecessors correspond to decision assignments.
- Special conflict vertices (κ) are added to the implication graph to indicate the occurrence of conflicts. The predecessors of a conflict vertex correspond to variable assignments that force a constraint ω to become unsatisfied. The directed edges are also all labeled with ω .

Figure 28.7 presents the implication graph produced after the assignment of value 1 to x_1 , assuming the set of constraints and truth assignments shown. The

Current Truth Assignment: $\{x_9 = 0@1, x_{12} = 1@2, x_{10} = 0@3, \dots\}$

Current Decision Assignment: $\{x_1 = 1@6\}$

- $\omega_1 : \overline{x_1} + x_2 + x_3 + x_9 \geq 2$
- $\omega_2 : 3\overline{x_4} + 2x_5 + 2x_6 + 4x_{10} \geq 4$
- $\omega_3 : 2\overline{x_2} + 3\overline{x_3} + 5x_4 \geq 5$
- $\omega_4 : \overline{x_5} + \overline{x_6} \geq 1$
- $\omega_5 : x_1 + x_7 + \overline{x_{12}} \geq 1$
- $\omega_6 : x_1 + x_8 \geq 1$
- $\omega_7 : \overline{x_7} + \overline{x_8} \geq 1$
- ...

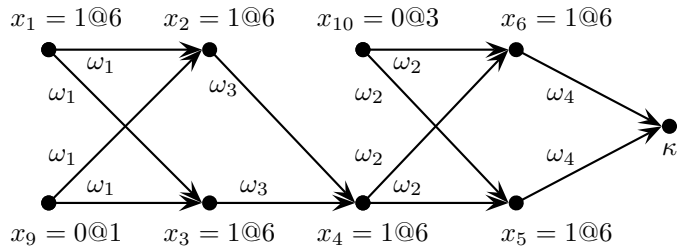


Figure 28.7. Example of an implication graph

implication graph contains a conflict node κ that shows a conflict on constraint ω_4 since both variables x_5 and x_6 are assigned value 1. Note that unlike the implication graph of SAT algorithms, in pseudo-Boolean formulations, the same constraint can imply several assignments.

When a logical conflict arises, the implication sequence leading to a conflict vertex κ is analyzed to determine the variable assignments that are responsible for the conflict. The conjunction of these assignments is an implicant for the conflict and represents a sufficient condition for the conflict to arise. Therefore, the negation of this implicant yields an implicate for the function associated with the problem constraints that must be satisfied. This new implicate is known as *conflict-induced clause*².

Let $A(\kappa)$ denote the conflicting assignment associated with a conflict vertex κ . The conflicting assignment is determined by a backward traversal of the implication graph starting at κ . Besides the decision assignment at the current decision level, only those assignments that occurred at previous decision levels are included in $A(\kappa)$. Hence, the current decision assignment with assignments from previous decision levels is a sufficient explanation for the conflict. For the example presented in Figure 28.7, we would have:

$$A(\kappa) = \{x_9 = 0@1, x_{10} = 0@3, x_1 = 1@6\} \tag{28.8}$$

The associated conflict-induced clause $\omega(\kappa)$ would then be:

$$\omega(\kappa) : x_9 + x_{10} + \overline{x_1} \geq 1 \tag{28.9}$$

²Conditions similar to these implicates are referred to as "no-goods" in Reasoning [dK86, SS77] and in some algorithms for Constraint Satisfaction Problems (CSP) [SV93].

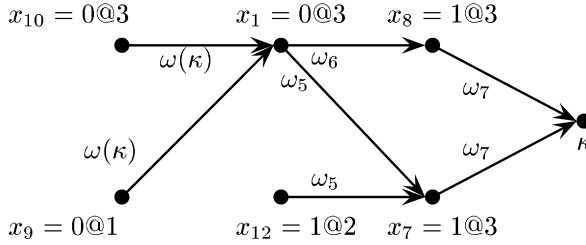


Figure 28.8. Implication graph after adding conflict-induced clause $\omega(\kappa)$ from (28.9)

After adding the new learned constraint $\omega(\kappa)$ to the list of the instance constraints to satisfy, the search backtracks to the decision level β determined by:

$$\beta = \max\{\delta(x_j) : (x_j = v(x_j)@ \delta(x_j)) \in A(\kappa) \wedge \delta(x_j) \neq \delta(\kappa)\} \tag{28.10}$$

where $\delta(\kappa)$ denotes the decision level of the conflict (current decision level). At decision level β either $\omega(\kappa)$ is used to imply a new assignment (the opposite value of the decision assignment at level β), or the conflict analysis procedure is executed again to determine a new backtracking decision level. In the example of Figure 28.7, the search would backtrack to decision level 3 and use $\omega(\kappa)$ to assert $x_1 = 0@3$. Observe that the assignment of value 0 to x_1 is not a decision assignment, but an implication from the conflict-induced clause. We refer to such assignments as *failure-driven assertions* (FDA) [MSS96], since they are implications of conflicts and not decision assignments.

Figure 28.8 shows the implication graph after adding the new learned constraint $\omega(\kappa)$ from (28.9). A new conflict arises and the conflict assignment $A(\kappa)$ associated with this conflict is:

$$A(\kappa) = \{x_9 = 0@1, x_{12} = 1@2, x_{10} = 0@3\} \tag{28.11}$$

Therefore, we would have a conflict-induced clause:

$$\omega(\kappa) : x_9 + x_{10} + \overline{x_{12}} \geq 1 \tag{28.12}$$

From (28.10) we can conclude that the search can backtrack directly to decision level 2 and propagate $x_{10} = 1@2$. This non-chronological backtrack avoids having to explore the opposite value of decision assignments made at intermediate decision levels.

28.6.3. Unique Implication Points

In an implication graph, a vertex a is said to *dominate* vertex b iff any path from the decision variable (or failure-driven assertion) of the decision level of a to b needs to go through a . A **Unique Implication Point** (UIP) [MSS96] is a vertex at the current decision level that dominates the conflict vertex κ in the implication graph at that decision level. For example, in Figure 28.7, vertex corresponding to the assignment $x_4 = 1$ dominates the conflict vertex κ . Hence, this vertex is an

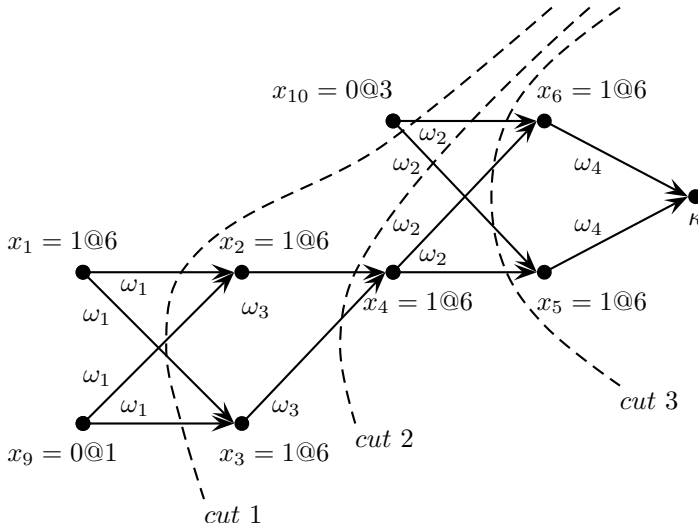


Figure 28.9. Cuts in implication graph

UIP in the implication graph. One should note that the vertex associated with the last decision assignment (or failure-driven assertion) is always a UIP.

The identification of UIPs in the implication graph can be used as a learning procedure [MSS96] by generating stronger implicates, i.e. with fewer literals than conflict-induced clauses. Considering that the UIP provides a single reason that implies the conflict on the current decision level, the same approach for generating conflict-induced clauses based on the last decision variable can be used for generating new clauses through a backward traversal of the implication graph until a UIP is found. Therefore, considering the UIP associated with assignment $x_4 = 1$ in Figure 28.7, a new clause can be added to the set of problem constraints

$$\omega_{uip} : \overline{x_4} + x_{10} \geq 1 \tag{28.13}$$

since the assignments $x_4 = 1$ and $x_{10} = 0$ are sufficient explanations for the conflict. Note that the same procedure can be applied to all UIPs found during the backward traversal of the implication graph, generating a new clause for each UIP.

It has been noted that the conflict-induced clause is generated by a bipartition of the implication graph [ZMMM01]. This bipartition is called a *cut* where the side containing the conflict vertex is called the *conflict partition* while the side with all vertices associated with decision assignments is called the *reason partition*.

In Figure 28.9 we show different cuts that can occur in the implication graph of Figure 28.7. The set of vertices on the reason partition that have at least one edge to the conflict partition provide an explanation for the conflict. Therefore, choosing a different cut of the implication graph corresponds to choose another explanation for the conflict and hence a different clause will be learned. The cuts

in Figure 28.9 would correspond to the following implicates:

$$\begin{aligned} \text{cut 1 : } \bar{x}_1 + x_9 + x_{10} &\geq 1 \\ \text{cut 2 : } \bar{x}_2 + \bar{x}_3 + x_{10} &\geq 1 \\ \text{cut 3 : } \bar{x}_4 + x_{10} &\geq 1 \end{aligned} \tag{28.14}$$

Observe that after backtracking both clauses generated from cut 1 or cut 3 are unit. Therefore, these clauses provide a mechanism to deduce an assignment to avoid the same conflict. However, the clause generated from cut 2 becomes unresolved after backtracking. The difference, is that both clauses from cut 1 and 3 are provided when the backward traversal of the implication graph is at an UIP [ZMMM01]. Hence, these clauses will always be unit after backtracking and provide the deduction of a failure-driven assertion. Several learning schemes can be used [ZMMM01], but current state of the art solvers usually stop at the first UIP when generating the conflict-induced clause since there is no need to make the complete backward traversal.

The conflict analysis approach presented in this section directly adapts to pseudo-Boolean solving the techniques developed for SAT solvers. Next, we describe the generalization of the conflict analysis procedure where the conflict-induced constraints are not necessarily propositional clauses. The purpose is to generate more expressive constraints than propositional clauses in order to not only provide a mechanism to deduce failure-driven assertions, but also to prune the search tree.

28.6.4. Generalizing Conflict Analysis

The purpose of applying a conflict analysis procedure is not only to be able to backtrack non-chronologically in the search tree, but also to learn a new constraint that implies a failure-driven assertion. In SAT algorithms, the new constraint is built using a sequence of resolution steps combining several propositional clauses. In fact, the propositional clauses selected for resolution operations are those labeling the edges used in a backward traversal of the implication graph [MSS96]. As described in section 28.6.3, the backward traversal can stop at the first UIP of the implication graph.

In section 28.5, cutting plane rules on pseudo-Boolean constraints were introduced. The new constraint resulting from the application of these rules is also known as a **cutting plane**. In fact, a cutting plane can be defined as any constraint that is implied by the original set of constraints [Hoo00]. However, as a result of the generalized resolution rule it is possible to obtain a weaker constraint. Hence, the final result from replacing resolution by the generalized resolution rule might not result in a conflict-induced constraint able to deduce a failure-driven assertion in a conflict analysis procedure.

Consider the partial assignment $\{x_2 = 0, x_4 = 1, x_5 = 0\}$ and suppose we have the following problem constraints:

$$\begin{aligned} \omega_1 : 3x_1 + 2x_2 + x_3 + 2\bar{x}_4 &\geq 3 \\ \omega_2 : 3\bar{x}_1 + x_5 + x_6 + x_7 &\geq 3 \end{aligned} \tag{28.15}$$

These constraints are in conflict since ω_1 implies $x_1 = 1$ while from ω_2 we must have $x_1 = 0$. If we apply generalized resolution to eliminate x_1 , we get the new constraint:

$$\omega_3 : 2x_2 + x_3 + 2\bar{x}_4 + x_5 + x_6 + x_7 \geq 3 \quad (28.16)$$

Note that the original constraints ω_1 and ω_2 were in conflict, but the resulting constraint ω_3 is not. This occurs due to the fact that ω_3 does not have the same information as the original constraints in (28.15) [Wil76]. Nevertheless, it has already been observed that generalized resolution can be strengthened with reduction and rounding operations so that no information is lost [Wil76].

In this section, an approach for building a conflict-induced pseudo-Boolean constraint that enables a failure-driven assertion is presented [CK03]. The idea is to maintain the conflict information in building the conflict-induced constraint while making the backward traversal of the implication graph. For that, at each step, it is ensured that the conflict-induced constraint being built remains unsatisfiable until the backtrack step takes place. After backtracking, the constraint becomes unit and is able to infer a failure-driven assertion. This can be achieved by performing reduction operations followed by saturation on both constraints before the generalized resolution operation [CK03]. However, reduction operations on constraints can be made only on unassigned literals or literals assigned value 1. In our example, weakening constraint ω_2 by reducing x_6 and x_7 (remember the weakening operation from section 28.5) we get:

$$\omega_2' : \bar{x}_1 + x_5 \geq 1 \quad (28.17)$$

Now, when performing generalized resolution between ω_1 and ω_2' to eliminate variable x_1 we have

$$\begin{array}{l} \omega_1 : 3x_1 + 2x_2 + x_3 + 2\bar{x}_4 \geq 3 \\ \omega_2' : 3(\bar{x}_1 + x_5 \geq 1) \\ \hline \omega_3' : 2x_2 + x_3 + 2\bar{x}_4 + 3x_5 \geq 3 \end{array} \quad (28.18)$$

Note that ω_3' is unsatisfied for the considered partial assignment $\{x_2 = 0, x_4 = 1, x_5 = 0\}$.

Algorithm 28.2 presents the pseudo-code for computing the conflict-induced pseudo-Boolean constraint $\omega(\kappa)$. Starting from the unsatisfied constraint c , each step generates a new cutting plane (**Cut_Resolve**) with the antecedent constraint of the variable assignment implied at the current decision level, but in reverse order of the Boolean constraint propagation. The procedure ends whenever the conflict-induced constraint $\omega(\kappa)$ implies an assignment at a previous decision level.

It must be observed that the constraint resulting from the generalized resolution might be oversatisfied, i.e., is not unsatisfied at the current decision level. To avoid this, procedures **Reduce1** and **Reduce2** must apply a series of reduction operations on literals that are unassigned or have value 1, such that the resulting constraint remains unsatisfied (see example (28.17) and (28.18)).

Although the learning of a general pseudo-Boolean constraint might enable a better pruning of the search tree, the data structures to manipulate these constraints have a higher computational overhead than the data structures for propositional clauses or cardinality constraints. Hence, it has been noted that in some

Algorithm 28.2 Generalized Conflict Analysis

V is the set of assigned variables at current decision level in nodes preceding κ in conflict graph sorted in reverse order of BCP

ω_i is the antecedent constraint of variable x_i

A_d is the set of assignments made on decision levels from 1 to d

```

1: function CONFLICT_ANALYSIS( $c, P$ )
2:    $\omega(\kappa) \leftarrow c$ 
3:   while  $V \neq \emptyset$  do
4:      $x_i \leftarrow \text{Remove\_Next}(V)$ 
5:      $\omega(\kappa) \leftarrow \text{Reduce1}(\omega(\kappa))$ 
6:      $\omega'_i \leftarrow \text{Reduce2}(\omega_i)$ 
7:      $\omega(\kappa) \leftarrow \text{Cut\_Resolve}(\omega(\kappa), \omega'_i, x_i)$ 
8:     if  $A_d$  triggers a literal implication in  $\omega(\kappa)$  then
9:        $\omega(\kappa) \leftarrow \text{Reduce3}(\omega(\kappa))$ 
10:      Add  $\omega(\kappa)$  to  $P$ 
11:      Backtrack to smallest  $d$  such that  $A_d$  implies a literal in  $\omega(\kappa)$ 
12:      return NO_CONFLICT
13:     end if
14:   end while
15:   return CONFLICT
16: end function

```

cases, learning propositional clauses or cardinality constraints might be more effective [CK03, CK05, SS05, SS06]. Algorithm 28.2 can be easily adapted to learn propositional clauses (as explained in the previous section) just by changing the reduction procedures to always reduce the constraints to propositional clauses. In order for the conflict-induced constraint to be a cardinality constraint, the procedure `Reduce3` can apply a cardinality reduction algorithm [CK03, CK05]. Note that the cardinality reduction algorithm must preserve the properties of the conflict-induced constraint to imply a failure-driven assertion. Although this step weakens the resulting conflict-induced constraint, the subsequent processing of cardinality constraints is usually faster.

Due to the overhead associated with learning pseudo-Boolean constraints, a hybrid learning scheme has been proposed [SS05, SS06]. The idea is to apply the propositional clause learning scheme (section 28.6.3) and the generalized learning scheme. Afterwards, the solver decides if it is worth to keep the general pseudo-Boolean constraint instead of the propositional clause based on how strong the pseudo-Boolean constraint is and on the number of literals with coefficients larger than 1 [SS05, SS06].

28.6.5. Optimization and Lower Bounding

In Pseudo-Boolean Optimization, the goal is to find a set of assignments to the problem variables such that all constraints are satisfied and the value of the cost function is minimized. One can easily extend a pseudo-Boolean solver to deal with optimization instances by doing a small modification to Algorithm 28.1 described in section 28.6. Instead of stopping when a solution to all constraints is found, one can determine the value of the current solution and add a new constraint such that a new solution must improve on the best solution found so far [Bar95a].

Algorithm 28.3 presents the pseudo-code for this approach. The idea is to

Algorithm 28.3 Generic Pseudo-Boolean Optimization Algorithm

```

1: function PBO_SOLVER( $P$ )
2:    $ub \leftarrow +\infty$ 
3:   while true do
4:     if Solution_Found( $P$ ) then
5:        $ub \leftarrow \sum_{j \in N} c_j v(x_j)$ 
6:       add constraint  $\sum_{j \in N} c_j x_j \leq ub - 1$  to  $P$ 
7:     else
8:       Decide( $P$ )
9:     end if
10:    while Deduce( $P$ )=CONFLICT do
11:      if Diagnose( $P$ )=CONFLICT then
12:        return  $ub$ 
13:      end if
14:    end while
15:  end while
16: end function

```

perform a linear search on the possible values of the cost function starting from the highest, at each step requiring the next computed solution to have a cost lower than the previous one. If the resulting instance is not satisfiable, then the solution is given by the last recorded solution. Notice that all techniques previously described for the decision problem can be readily used for optimization instances.

Nevertheless, the classical approach for solving combinatorial optimization problems using backtrack search to exhaustively explore the search space is Branch and Bound [LW66]. In this approach, an **upper bound** on the value of the cost function is maintained and updated whenever a lower cost solution to the problem constraints is found. Additionally, at each node of the search tree, a **lower bound** on the value of the cost function is estimated considering the current partial assignment to problem variables. Whenever the lower bound estimate is higher than or equal to the upper bound, the search can be pruned since no better solution can be found by extending the current partial assignment. Hence, for optimization instances, the use of lower bound estimates can be a valuable pruning technique.

When solving a Pseudo-Boolean Optimization instance P , let $P.path$ denote the total cost of the partial assignment at some node in the search tree. Let $P.lower$ denote the lower bound estimate on the value of the cost function in order to satisfy all unresolved constraints at the same node. Clearly, at this node of the search tree, the lower bound value is given by the sum of $P.path$ and $P.lower$. Additionally, let $P.upper$ denote the cost of the best solution found so far. It is said that a **bound conflict** occurs when

$$P.path + P.lower \geq P.upper \quad (28.19)$$

in this case, a better solution than $P.upper$ cannot be found with the current partial assignment and the search can be pruned.

The most commonly used procedures to determine the $P.lower$ are based on finding a Maximum Independent Set of constraints [CM95] and Linear Programming Relaxation [LD97]. However, other methods can also be used namely

Lagrangian relaxation [AMO93] or the Log-approximation procedure [Cou96] initially proposed for Unate and Binate Covering Problems.

The effectiveness of lower bounding is usually determined by how tight the estimate is with respect to the optimum solution. The motivation is that with higher lower bound estimates, large portions of the search space can be pruned earlier. It has been shown that the Log-approximation procedure is able to determine a lower bound estimate that is log-approximable [Cou96] to the optimum value of the cost function, but it is dependent on being able to find a solution using a greedy algorithm. Although that is easy when solving Unate Covering instances, that is not the case for general case of Pseudo-Boolean Optimization.

28.6.5.1. Maximum Independent Set

The **maximum independent set** lower bounding procedure is a greedy method based on identifying an independent set of unresolved constraints. The objective of the greedy procedure is to determine a set of unresolved constraints with no literals in common such that the minimum cost to satisfy each constraint is non-zero.

First, in order to build a maximum independent set M , it is necessary to determine the minimum cost necessary to satisfy each unresolved constraint. Remember that a pseudo-Boolean constraint ω_i can be viewed as:

$$\omega_i = \sum_{j \in N} a_{i,j} l_j \geq b_i, \quad (28.20)$$

Suppose all no-cost literals from ω_i are true and let s_{nc} be the sum of the coefficients of all no-cost literals from ω_i . Let ω'_i be defined as:

$$\omega'_i = \sum_{j \in C} a_{i,j} x_j \geq b_i - s_{nc}, \quad (28.21)$$

where C is the set of literals from ω_i with positive cost. Finding the minimum cost for satisfying ω'_i provides the minimum cost to satisfy ω_i and that can be defined as:

$$\begin{aligned} & \text{minimize} && \sum_{j \in C} c_j \cdot x_j \\ & \text{subject to} && \omega'_i \end{aligned} \quad (28.22)$$

Observe that (28.22) is a special case of a PBO problem known as *knapsack 0-1* problem. Nevertheless, it is still an NP-Complete problem [Kar72] and it can be solved in pseudo-polynomial time using dynamic programming [Dan57].

Alternatively, one can use an approximation algorithm for the problem of finding the minimum cost of satisfying a pseudo-Boolean constraint by using a greedy algorithm. First, determine the minimum number of literals that need to be true in order to satisfy ω'_i by reducing ω'_i to a cardinality constraint [Bar95b]. This can be done by summing the sorted set of ω'_i coefficients, starting from the largest $a_{i,j}$ and checking if the sum of $\sum_{j=1}^k a_{i,j} < b_i - s_{nc}$ holds for each value of k . A more detailed description of the cardinality reduction algorithm can be found in [CK03].

Suppose that ω'_{ic} denotes the cardinality constraint obtained by the cardinality reduction algorithm applied to ω'_i

$$\omega'_{ic} = \sum_{j \in C} x_j \geq k \quad (28.23)$$

a lower bound on the minimum cost to satisfy ω'_i is given by accumulating the cost of the first k literals in a sorted set of literal coefficients in the problem cost function, starting with the lowest c_j .

Building the maximum independent set of constraints M is done by adding, at each step, a new constraint with no literal in common with any other constraint already in M . The minimum cost for satisfying the set of constraints M is a *lower bound* on the solution of the problem instance and is given by,

$$Cost(M) = \sum_{\omega_i \in M} MinCost(\omega_i) \quad (28.24)$$

where $MinCost(\omega_i)$ denotes the lower bound on the minimum cost to satisfy ω_i given by the approximation algorithm described in this section.

Although the lower bound provided by the maximum independent set of constraints can be arbitrarily far from the optimum, it is a simple and fast method to determine a lower bound on the problem instance. Linear programming relaxations [LD97] usually find tighter bounds, but are much more time consuming, in particular on highly constrained instances.

Some pruning techniques can be applied after the lower bound estimation when using the maximum independent set of constraints. One of these techniques is the *limit lower bound* first proposed in [CM95] which states that one can also prune the search tree by identifying some necessary assignments. That occurs whenever after calculating the lower bound estimation the following is true:

$$P.path + P.lower + Cost(x_j) \geq P.upper \quad (28.25)$$

when x_j is unassigned, but does not occur in any constraint $\omega_i \in M$, then $x_j = 0$ is a necessary assignment since otherwise a bound conflict would occur.

28.6.5.2. Linear Programming Relaxation

Although the approximation of a maximum independent set of constraints (MIS) is the most widely used lower bound procedure for the Binat Covering problem [Cou96, VKBSV97], **linear programming relaxation** (LPR) has also been used with success [LD97]. It is also often the case that the bound obtained with linear programming relaxation is higher than the one obtained with the MIS approach [LD97]. Moreover, linear programming relaxations have long been used for lower bounding estimation in branch and bound algorithms for solving integer linear programming problems [NW88].

The general formulation of the LPR for a pseudo-Boolean problem is obtained

from the original PBO instance as follows:

$$\begin{aligned}
 & \text{minimize} && z_{lpr} = \sum_{j=1}^n c_j \cdot x_j \\
 & \text{subject to} && \sum_{j=1}^n a_{i,j} x_j \geq b_i \\
 & && 0 \leq x_j \leq 1
 \end{aligned} \tag{28.26}$$

where c_j define the integer cost associated with every decision variable x_j , $a_{i,j}$ define the variable coefficients and b_i the right-hand side of every constraint. Notice that in the LPR formulation the decision variables are no longer Boolean values and can take a non-integer value between 0 and 1.

Let z_{lpr}^* denote the optimum value of the cost function of the LPR (28.26) and z_{pb}^* denote the optimum value of the cost function of the PBO instance. It is well-known that z_{lpr}^* is a lower bound on z_{pb}^* [NW88]. Basically, any solution of the original PBO problem is also a feasible solution of (28.26), but the converse is not true. Moreover, if the solution of (28.26) is integral (i.e. for all variables we have $x_j \in \{0, 1\}$), then we necessarily have $z_{pb}^* = z_{lpr}^*$. Furthermore, different linear programming algorithms can be used for solving (28.26), some with guaranteed worst-case polynomial run time [NW88].

In classical branch and bound algorithms, backtracking due to bound conflicts corresponds to a chronological backtrack in the search tree. However, current state-of-the-art PBO solvers are based on conflict-driven constraint learning algorithms. Therefore, whenever a bound conflict occurs, a new unsatisfied constraint must be added such that it *explains* the bound conflict. Since the explanation is represented as an unsatisfied constraint, it can be treated by the solver in the same way as a logical conflict. Hence, a conflict analysis procedure is carried out on the new constraint and the algorithm safely backtracks in the search tree. In order to establish an explanation for a bound conflict, it is necessary to take into account which lower bounding procedure is being used. Procedures to determine explanations for bound conflicts using the Maximum Independent Set [MMS04], Linear Programming Relaxations [MMS04] and Lagrangian Relaxations [MMS05] have already been established.

28.6.6. Cutting Planes

There are several techniques for pruning the search tree in PBO algorithms. One of the most well-known is the generation of cutting planes that are supposed to be able to readily exclude partial assignments that cannot be extended into a full assignment corresponding to a model [Hoo00]. However, these new cutting planes cannot exclude any model from the original pseudo-Boolean problem. In the context of optimization problems, the importance of cutting planes is on the ability to strengthen relaxations. In this section, the focus will be on the relation between Linear Programming Relaxation and cutting planes for PBO and refer to Hooker's work [Hoo00] for reference on cutting planes techniques and analysis.

Cutting plane techniques were first introduced for Integer Linear Programming and work on cutting planes can be traced back to Gomory [Gom58]. The original idea of Gomory's cutting planes was to derive new linear inequalities in

order to exclude some non-integer solutions when performing Linear Programming Relaxation (LPR) for lower bound estimation. However, the new linear inequalities are valid for the original integer linear program and so can be safely added to the original problem. Moreover, the LPR with the added inequalities may yield a tighter lower bound estimate. In spite of being first developed for Integer Linear Programming, Gomory's cutting planes can be readily used in Pseudo-Boolean Optimization.

Section 28.6.5.2 describes the utilization of linear programming relaxation (LPR) for estimating lower bounds in PBO. In simplex-based solutions for solving the LPR, the simplex method adds a set S of slack variables (one for each constraint) such that,

$$\begin{aligned} \text{minimize} \quad & z_{lpr} = \sum_{j=1}^n c_j \cdot x_j \\ \text{subject to} \quad & \sum_{j=1}^n a_{i,j} x_j - s_i = b_i \\ & 0 \leq x_j \leq 1, s_i \geq 0 \end{aligned} \tag{28.27}$$

This formulation is called the slack formulation and it is used to create the original simplex tableau [NW88].

Let x^* denote the optimal solution of the LPR. If the solution x^* is integral, then x^* provides the optimal solution to the original problem. Otherwise, in order to generate a new cutting plane, Gomory suggests to choose a basic³ variable x_j such that its value on the LPR solution is not integral. Since x_j is a basic variable, after the pivot operations performed by the simplex algorithm on (28.27), there is a row in the simplex tableau of the form,

$$x_j + \sum_{i \in P} \alpha_i x_i + \sum_{i \in Q} \beta_i s_i = x_j^* \tag{28.28}$$

where P and Q are the sets of indexes of non-basic variables (problem variables and slack variables, respectively). Gomory [Gom58] proves that the inequality,

$$\begin{aligned} \sum_{i \in P} f(\alpha_i) x_i + \sum_{i \in Q} f(\beta_i) s_i &\geq f(x_j^*) \\ \text{where } f(y) &= y - \lfloor y \rfloor, y \in \mathbb{R} \end{aligned} \tag{28.29}$$

is violated by the solution of the LPR, but satisfied by all non-negative integer solutions to (28.28). Hence, it is also satisfied by all solutions to the original PBO problem and can be added to the LPR. Solving the LPR with the new restriction (known as a Gomory *fractional cut*) will yield a tighter lower bound estimate on the value of the objective function.

Gomory fractional cuts were the first cutting planes proposed for Integer Linear Programming. However, several methods for strengthening these cuts have been proposed [BCCN96, BJ80, CCD95, Gom60, LL02]. In fact, Gomory [Gom60]

³See for example [NW88] for a definition of basic and non-basic variables.

himself proves that the cut

$$\sum_{i \in P} g(\alpha_i)x_i + \sum_{i \in Q} g(\beta_i)s_i \geq 1$$

$$\text{where } g(y) = \begin{cases} \frac{f(y)}{f(x_j^*)} & : f(y) \leq f(x_j^*) \\ \frac{1-f(y)}{1-f(x_j^*)} & : f(y) > f(x_j^*) \end{cases} \quad (28.30)$$

is stronger than (28.29) and satisfied by all solutions of (28.27). These cuts are known as Gomory *mixed-integer cuts*.

One should note that from (28.27) each slack variable depends only from the original problem variables and can be replaced in (28.29) by $s_i = \sum_{j \in N} a_{i,j}x_j - b_i$. Afterwards, if we apply the rounding operation on the non-integer coefficients we obtain a new pseudo-Boolean constraint valid for the original PBO instance, since the rounding operation will only weaken the constraint.

Since Gomory's original work, a large number of cutting plane techniques have been proposed [Bix94, Chv73, NW88]. For example, the Chvátal-Gomory cuts [Chv73] is another approach to produce cutting planes where the slack variables are not necessary. Chvátal proves that any Gomory cut can be generated using his approach by producing a new inequality through a weighted combination of inequalities (generalized resolution rule) followed by rounding. However, unlike Gomory, he does not present a systematic way for obtaining useful cuts.

In a modern Conflict-Driven algorithm, a conflict analysis procedure is carried out whenever a conflict arises [MSS96, MMZ⁺01]. Therefore, if a generated cutting plane is involved in the conflict analysis process, the algorithm must be able to determine the correct logical dependencies in order to backtrack to a valid node of the search tree. In order to do that, it is necessary to associate dependencies to the computed cutting plane [MMS06], thus enabling constraint learning and non-chronological backtracking from constraints inferred with cutting plane techniques.

28.6.7. Translation to SAT

Pseudo-Boolean constraints can be **translated to SAT** and surprisingly this approach can be quite efficient as first demonstrated by the minisat+ solver [ES06] in several evaluations of pseudo-Boolean solvers [MR07].

28.6.7.1. Translation of constraints

A naive approach is to translate a PB constraint as a set of clauses which is semantically equivalent and uses the same variables. Unfortunately, an exponential number of clauses can be generated (as is the case with cardinality constraints) and therefore this approach is unusable in the general case.

Other approaches will introduce extra variables to generate a short clausal formula F which is semantically equivalent to the PB formula P in an extended sense (all models of F restricted to the vocabulary of P are models of P and conversely, models of P can be extended to obtain models of F).

A very basic approach is to translate each PB constraint as a sequence of binary adders followed by a comparator. Let $\sum_j a_j x_j \geq b$ be the PB constraint to

translate. The first adder will sum the first two terms a_1x_1 and a_2x_2 to get partial sum S_2 . Successive adders will sum S_{i-1} and a_jx_j to obtain S_j . Notice that a_jx_j is simply obtained by replacing each 1 in the binary representation of a_j by the x_j variable. The last sum S_n is compared to b by a final comparator. The output of this comparator is constrained so that an inconsistency is generated when $S_n \geq b$ does not hold. A slightly refined version of this basic idea is detailed in [War98]. These adders and comparator can be encoded linearly as clauses by introducing extra variables and this encoding is obviously equivalent to the PB constraint. However, it does not let unit propagation maintain generalized arc consistency which means that unit propagation will not be able to infer that a variable must be assigned a value v whereas this is logically implied by the assignment of the other variables. The problem with this encoding lies in the sequence of adders. As soon as one single variable x_j is unassigned, each following partial sum is also undefined and the comparator cannot enforce any constraint. As an example, with constraint $x_1 + x_2 + x_3 + x_4 \geq 3$ and current assignment $x_1 = x_2 = 0$, no inconsistency will be detected until x_3 and x_4 are also assigned. This is of course a terrible source of inefficiency for a solver. On the same example, an encoding that would let unit propagation enforce generalized arc consistency would be able to infer that x_2, x_3 and x_4 must all be assigned true as soon as x_1 is assigned false.

To have unit propagation automatically enforce generalized arc consistency, a BDD-like (Binary Decision Diagram described in Part 1, Chapter 3) encoding may be used [ES06, BBR06]. The root of the BDD represents the constraint to encode: $\sum_{j=1}^n a_jx_j \geq b$. A node in the BDD will represent a constraint $\omega_{m,c}$ defined by $\sum_{j=m}^n a_jx_j \geq c$. A propositional variable $c_{m,c}$ is associated to each node to indicate if $\omega_{m,c}$ is satisfied. There are two possibilities to satisfy $\omega_{m,c}$ defined by $\sum_{j=m}^n a_jx_j \geq c$. The first possibility is that $x_m = 1$ and $\sum_{j=m+1}^n a_jx_j \geq c'$ with $c' = c - a_m$. The second one is that $x_m = 0$ and $\sum_{j=m+1}^n a_jx_j \geq c'$ with $c' = c$. Therefore, each node $\omega_{m,c}$ in the BDD will have at most two children: $\omega_{m+1,c-a_m}$ and $\omega_{m+1,c}$. Some nodes may have less than two children because some constraints are trivially false ($\omega_{m,c}$ for any $c > \sum_{j=m}^n a_j$) or trivially true ($\omega_{m,0}$ for any m) and the BDD is simplified. Besides, two nodes may share a child. For example, if $a_m = 1$ and $a_{m+1} = 1$, node $\omega_{m,c}$ will have two children $\omega_{m+1,b}$ and $\omega_{m+1,b-1}$ which will share a common child $\omega_{m+2,b-1}$. The relation between the truth of a node and the truth of its children is encoded in a few clauses which only depend on x_m and the propositional variables associated to both nodes. [ES06] used Tseitin transformation of BDD and [BBR06] uses a slightly different encoding. These encodings let unit-propagation maintain generalized arc consistency of the pseudo-Boolean constraint. Unfortunately, it generates encodings of exponential size for some constraints [BBR06].

The third encoding is founded on the unary representation of numbers [BB03, ES06]. In unary notation, a number between 0 and m will be represented by exactly m propositional variables. A number n is represented by exactly n variables set to true and all other variables set to false (the value of the number is the number of true variables). To represent that the left side of a constraint is currently comprised in $[a, b]$, a variables will be set to true, $m - b$ variables are set to false and the $b - a$ remaining variables are unassigned. Sorting networks can be

used and encoded as clauses to make sure that a number X will be represented by propositional variables x_1 to x_m such that each true variable precedes any unassigned variable and each unassigned variable comes before any false variable, i.e. $\exists a, b$ s.t. $\forall i \leq a, x_i = 1, \forall a < i \leq b, x_i$ is unassigned, $\forall i > b, x_i = 0$. With this normalized unary representation, the constraint $X \geq b$ is encoded by $x_b = 1$ and $X < b$ is encoded by $x_b = 0$ which are easily handled by a SAT solver. Unfortunately, unary representation of numbers is generally too large to be used in practice. [ES06] introduces another representation of numbers in a base represented by a sequence of positive integers denoted $\langle B_0, B_1, \dots, B_{n-1} \rangle$. Each B_i is a constant and $B_0 = 1$. A number is represented by a sequence of digits $\langle d_0, d_1, \dots, d_{n-1}, d_n \rangle$ such that $0 \leq d_i < B_i$ for $i < n$. The notation $\langle d_0, d_1, \dots, d_{n-1}, d_n \rangle$ represents the number $\sum_{i=0}^n d_i \cdot \prod_{j=0}^i B_j$. This representation is a generalization of the usual representation of numbers in a base B with the exception that the ratio of the weights of two successive digits is not a constant. In [ES06], each digit d_i is represented in unary notation. Therefore a number is represented by several blocks in unary notation and carries are used to propagate information between blocks. The strength of this encoding is that it is a drastic compression of the unary representation.

The solver minisat+ [ES06] chooses between these different representations (adder, BDD or unary-based representation) the most appropriate to encode each individual PB constraint.

28.6.7.2. Polynomial PB Encodings enforcing GAC by unit propagation

Until 2009, SAT encodings of PB constraints either did not allow unit propagation (UP) to maintain generalized arc consistency (GAC), or were exponential in the worst case. The first encoding which lets UP maintain GAC was proposed in [BBR09]. This encoding was not designed for efficiency, but rather as a proof of concept. We consider *less than* constraints $\sum_{j=1}^n a_j l_j < b$ that are non trivial ($\sum_{j=1}^n a_j \geq b$), and denote by $p + 1$ the number of bits required to encode the numbers a_j . A first step is to let unit propagation detect when this constraint is falsified. To this end, each coefficient is written in binary: $a_j = \sum_{i=0}^p a_j^i \cdot 2^i$. Next, to facilitate the comparison between the left and right side of the constraint, we add on both sides a constant t (the tare, which is also written in binary $t = \sum_{i=0}^p t^i 2^i$) such that $b + t$ is the smallest multiple of 2^p . Let m be the smallest integer such that $b + t = m 2^p$. The constraint is then written as $\sum_{i=0}^p 2^i (t^i + \sum_{j=1}^n a_j^i l_j) < m \cdot 2^p$. Terms with weight 2^i are grouped into a bucket $B_i = \langle t^i, a_0^i l_0, a_1^i l_1, \dots, a_n^i l_n \rangle$. These buckets only contain constants and literals and are easily sorted by unit propagation (by encoding a sorting network or a totalizer [BB03]) such that literals or constant set to 1 appear first in the bucket, unassigned literals appear next and literals or constant set to 0 appear last. Let B'_i be the sorted bucket corresponding to B_i . B'_i encodes in unary notation the current interval of possible values for the terms with weight 2^i . When bucket B'_i contains two consecutive 1s, a carry must be propagated to bucket B_{i+1} . To this end, we define the half of a sorted bucket $B'_i = \langle x_0, x_1, \dots, x_n \rangle$ as the bucket containing only variables with odd indexes: $\frac{1}{2} B'_i = \langle x_1, x_3, x_5, \dots, x_{2k+1}, \dots \rangle$. We can now redefine B_i as $\langle t^i, a_0^i l_0, a_1^i l_1, \dots, a_n^i l_n \rangle + \frac{1}{2} B'_{i-1}$ where $+$ on buckets represents both concatenation of contents and addition of the associated values, and

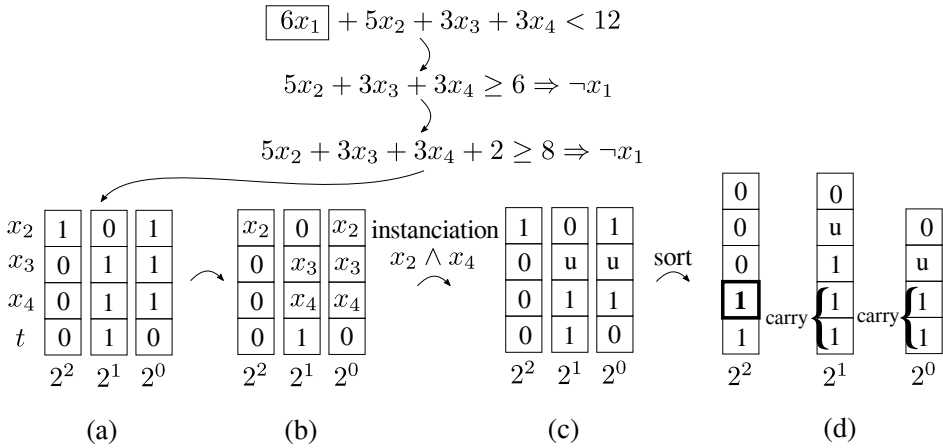


Figure 28.10. Example of propagation on the [BBR09] encoding. Given the constraint $6x_1 + 5x_2 + 3x_3 + 3x_4 < 12$, the example presents how it is detected if x_1 must be set to 0. Subfigure (a) presents the coefficients in binary and subfigure (b) presents the multiplication. Given an assignment $x_2 \wedge x_4$, subfigure (c) shows the instantiation of variables. Subfigure (d) presents the result after sorting and carry propagation. The 1 shown in bold (the m -th term of the last bucket) indicates that x_1 must be set to 0, because the left side of the bottom constraint is at least 8.

by convention B_{-1} is the empty bucket. With all this setup, a bucket B_i now represents in unary notation the sum of all terms with weight 2^j such that $j \leq i$. The constraint is falsified if and only if the left side of the rewritten constraint is equal to $m2^p$, which is true if and only if the m -th term of bucket B'_p is true, and this can be computed by unit propagation. This means that we can enforce the satisfaction on the constraint by setting the m -th term to 0. This encoding uses $O(n \log(n) \log(a_{max}))$ variables and $O(n^2 \log(n) \log(a_{max}))$ clauses, where a_{max} is the largest coefficient of the constraint.

The last step to enforce GAC with unit propagation is to consider each term of the constraint individually. As soon as $\sum_{j=1, j \neq i}^n a_j l_j \geq b - a_i$, l_i is necessarily false. In other words, l_i must be false whenever $\sum_{j=1, j \neq i}^n a_j l_j < b - a_i$ is falsified, which is detected by the above encoding when the m -th term of the last bucket is true. So for each literal l_i , we just have to encode $\sum_{j=1, j \neq i}^n a_j l_j < b - a_i$, identify the m -th term (x_m) of the last bucket and add the implication $x_m \Rightarrow \neg l_i$. This is sufficient to let unit propagation maintain GAC. Figure 28.10 illustrates unit propagation on this encoding.

28.6.8. Dealing with the Optimization Problem

Algorithm 28.3 presented in section 28.6.5 describes a generic approach for solving Pseudo-Boolean Optimization. In this case, a linear search on the possible values of the objective function is performed. The objective function $c(x_1, \dots, x_n)$ to be minimized is translated to a constraint $c(x_1, \dots, x_n) \leq ub - 1$ where ub is an inte-

Algorithm 28.4 Linear Search SAT-UNSAT

```

1: function LINEARSEARCH-SAT-UNSAT( $P, c$ )
2:    $ub \leftarrow \sum_{j \in N} c_j + 1$ 
3:   while true do
4:     if PBS( $P$ ) = SATISFIABLE then
5:        $ub \leftarrow \sum_{j \in N} c_j v(x_j)$ 
6:        $P \leftarrow P \cup \{\sum_{j \in N} c_j x_j \leq ub - 1\}$ 
7:     else
8:       return  $ub$ 
9:     end if
10:  end while
11: end function

```

ger which is modified at each iteration of the algorithm. Initially, ub is initialized with a value higher than the maximum value of the objective function. Each time a solution S is found, ub is set to $c(S)$ so that the constraint corresponding to the objective function imposes to find a strictly better solution than S . This process ends when the formula becomes unsatisfiable. Therefore, the last solution (if it exists) is an optimal solution.

The linear search approach can be viewed as a sequence of iterative calls to a Pseudo-Boolean Satisfiability (PBS) oracle that determines if a given Pseudo-Boolean formula P is satisfiable. Algorithm 28.4 summarizes this approach.

It should be noted that enforcing the constraint $c(x_1, \dots, x_n) \leq ub$ at each step of this algorithm does not necessarily require adding a new constraint to the formula. Some extra Boolean variables b_j can be introduced so that the single constraint $c(x_1, \dots, x_n) \leq ub - \sum_j 2^j b_j$ is introduced at the beginning of the algorithm. Initially, each b_j is set to false and for each new solution S , these variables are instantiated so that $ub - \sum_j 2^j b_j = c(S) - 1$. Furthermore, observe that the learned constraints in the PBS oracle remain valid between iterations. Hence, learned constraints from previous PBS calls can be used to cut the search space.

In the binary search approach, the objective function $c(x_1, \dots, x_n)$ is translated to the two constraints $lb \leq c(x_1, \dots, x_n)$ and $c(x_1, \dots, x_n) \leq ub$ where lb and ub are integers which are modified at each iteration of the algorithm. Initially, lb is set to the minimum value of the objective function (lower bound) and ub is set to a value higher than the maximum value of the objective function (upper bound). At each step, the middle M of interval $[lb, ub]$ is computed and solutions are searched in the interval $[lb, M]$. If a solution S is found, the search continues in the interval $[lb, c(S) - 1]$. Otherwise, the search is performed in the interval $[M + 1, ub]$ (this is mostly the usual binary search algorithm). Algorithm 28.5 summarizes this process using a PBS oracle. The last solution found by this algorithm (if any) is an optimal solution. This algorithm is known to converge faster toward the optimal solution, but will usually spend much more time than the linear search to find the first solutions. Moreover, when there is an unsatisfiable call to the PBS oracle, any learnt constraint which was inferred from the search bounds must be forgotten from one call of the PBS oracle to another, which may be difficult to implement.

Algorithm 28.5 Binary Search

```

1: function BINARYSEARCH( $P, c$ )
2:    $lb \leftarrow 0$ 
3:    $ub \leftarrow \sum_{j \in N} c_j + 1$ 
4:   while  $lb < ub$  do
5:      $M \leftarrow \frac{(lb+ub)}{2}$ 
6:     if  $\text{PBS}(P \cup \{lb \leq \sum_{j \in N} c_j x_j \leq M\}) = \text{SATISFIABLE}$  then
7:        $ub \leftarrow \sum_{j \in N} c_j v(x_j)$ 
8:     else
9:        $lb \leftarrow M + 1$ 
10:    end if
11:  end while
12:  return  $ub$ 
13: end function

```

28.6.8.1. Core-guided algorithms

Algorithm 28.4 is said to be a SAT-UNSAT linear search since the first calls to the PBS oracle are satisfiable, while the last call is unsatisfiable. However, one can also perform an UNSAT-SAT linear search on the value of the objective function. In this case, we define a lower bound lb (initially set to 0) and at each iteration a constraint $c(x_1, \dots, x_n) \leq lb$ is considered. If the PBS oracle call is unsatisfiable, then the lower bound is incremented and the constraint is replaced considering the new value for lb . The process is repeated until a satisfiable call is reached. When a satisfiable call is obtained, then an optimal solution was found, since all lower values of the objective function were already tried.

Observe that in this UNSAT-SAT algorithm, at each iteration, learned constraints that depend on the objective function constraint must be discarded. Moreover, all non-zero coefficients in the objective function must be considered in the constraint that limits the value of the objective function.

In the last decade, a plethora of new algorithms for Maximum Satisfiability (MaxSAT) have been proposed that resulted in a dramatic progress in MaxSAT solver performance for many real-world applications. Most of these algorithms perform an UNSAT-SAT search, but take advantage of the ability of SAT solvers to be able to identify a subset of clauses that are unsatisfiable (also known as unsatisfiable cores). Despite being proposed to solve MaxSAT, core-guided algorithms can also be used to solve Pseudo-Boolean Optimization.

There are several variants of MaxSAT. However, in the context of this chapter, MaxSAT refers to the more general weighted partial MaxSAT problem. Let (F_H, F_S) define a weighted partial MaxSAT problem such that F_H denotes the set of hard clauses and F_S denotes the set of soft clauses. For each soft clause ω_i there is a positive weight c_i that represents the cost of not satisfying ω_i . Given a MaxSAT instance (F_H, F_S) , the goal is to find an assignment such that it satisfies all hard clauses in F_H and minimizes the sum of the weights of unsatisfied soft clauses in F_S .

Weighted Boolean Optimization (WBO) is a generalization of MaxSAT. A WBO instance (F_H, F_S) is defined in the same way as MaxSAT, but hard constraints in F_H and soft constraints in F_S can be pseudo-Boolean constraints [MMSP09]. Given a pseudo-Boolean Optimization instance defined over a set

Algorithm 28.6 MSU3 for Weighted Boolean Optimization

```

1: function MSU3( $(F_H, F_S)$ )
2:    $lb \leftarrow 0$ 
3:    $V_R \leftarrow \emptyset$ 
4:    $P \leftarrow F_H \cup F_S$ 
5:   while true do
6:      $(st, v, C) \leftarrow \text{PBS}(P \cup \{\sum_{r_j \in V_R} c_j r_j \leq lb\})$ 
7:     if  $st = \text{SATISFIABLE}$  then
8:       return  $\sum_{r_j \in V_R} c_j v(r_j)$ 
9:     end if
10:    for all  $\omega_j \in (C \cap F_S)$  do
11:       $V_R \leftarrow V_R \cup \{r_j\}$ 
12:       $\omega_R \leftarrow \omega_j \vee r_j$ 
13:       $P \leftarrow (P \setminus \{\omega_j\}) \cup \{\omega_R\}$ 
14:    end for
15:     $lb \leftarrow lb + 1$ 
16:  end while
17: end function

```

of constraints P and an objective function $\sum_{j \in N} c_j x_j$, one can build the corresponding WBO (F_H, F_S) instance as follows: (i) define $F_H = P$ as the hard constraints and (ii) for each term $c_j x_j$ in the objective function, add $\bar{x}_j \geq 1$ with a weight of c_j to F_S . Clearly, an optimal solution of the WBO instance is also a solution of the pseudo-Boolean Optimization instance.

MaxSAT algorithms based on iterative calls to a SAT solver can be easily adapted to solve WBO (and pseudo-Boolean Optimization as a result). One just needs to replace each call to a SAT solver with a call to a PBS solver. However, in order to use a core-guided algorithm, the PBS solver to use must be able to identify an unsatisfiable core of the formula.

As an example of adaptation of MaxSAT algorithms, a generalization of the MSU3 [MP07] algorithm is presented to solve a WBO instance (F_H, F_S) . Algorithm 28.6 follows a core-guided linear UNSAT-SAT search. First, the algorithm starts by initializing the lower bound to 0 and building a pseudo-Boolean satisfiability formula joining both hard and soft constraints (line 4). Next, an iterative process starts by calling a PBS oracle (line 6). Note that at each call of the PBS oracle, a constraint is considered such that the total weight of unsatisfied soft constraints is not larger than lb .

In this algorithm, the PBS oracle returns a triple (st, v, C) where st denotes if the formula is satisfiable or unsatisfiable, v contains an assignment if the formula is satisfiable, and C contains an unsatisfiable core if the formula is unsatisfiable. Whenever the PBS oracle returns unsatisfiable, for each soft constraint ω_j in the unsatisfiable core C , a new relaxation variable r_j is created and added to V_R . The cost c_j of unsatisfying soft constraint ω_j is associated with variable r_j . Next, a new constraint ω_R is build such that r_j must be assigned to true if the original soft constraint ω_j is not satisfied (line 12), and ω_j is replaced by ω_R in P (line 13). Moreover, at each iteration, the lower bound is incremented ⁴ Finally, the algorithm ends when the PBS oracle returns a satisfiable assignment that

⁴Depending on the values for the weights of soft constraints, the lower bound can be increased by more than one [MHL⁺13].

corresponds to an optimal solution.

We refer to Chapter 23 and 24 for an extensive explanation of core-guided algorithms, as well as further details on these algorithms, such as different techniques of incremental implementation. Nevertheless, it should be noted that core-guided algorithms can also be adapted to pseudo-Boolean optimization (as exemplified in this section) or other optimization problems such as Maximum Satisfiability Modulo theories [BN15]. Moreover, many core-guided and linear search algorithms for MaxSAT rely on efficient usage of cardinality and pseudo-Boolean constraints. In fact, several current state of the art algorithms for MaxSAT rely on effective translations of cardinality and pseudo-Boolean constraints.

28.7. Conclusion

Pseudo-Boolean constraints offer a rich framework for knowledge representation and inference and still remain close enough to the well-known SAT problem and the integer linear programming paradigm to benefit from the experience in both fields. Pseudo-Boolean constraints are more expressive than clauses and optimization problems can be naturally expressed in this formalism. Proofs in the pseudo-Boolean formalism can be exponentially shorter than in the clausal formalism, and pseudo-Boolean encodings can be exponentially shorter than clausal encodings.

Pseudo-Boolean solvers integrate techniques from both the SAT and ILP community. Integer Linear Programming techniques usually focus on identifying the optimal non-integer solution and then will try to identify integral solutions close to this optimal non-integer solution. In contrast, SAT techniques focus on identifying integral solution and search for the optimal solution among them. The integration of these techniques is undoubtedly a promising method to solve a variety of problems (both decision and optimization problems).

At this time, it is hard to identify one kind of algorithm among the existing solvers as the most efficient to solve pseudo-Boolean problems. The pseudo-Boolean field has not yet reached the same maturity as SAT solving and a number of major improvements can be expected in the coming years. Solving pseudo-Boolean formulas raises new challenges, both from a theoretical and implementation point of view. Trade-offs between inference power and inference speed are often made in current algorithms and the right balance is still sought. We can expect that, once the right balance is found, pseudo-Boolean solvers will become a major tool in problem solving.

References

- [AMO93] R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Pearson Education, 1993.
- [Bar95a] P. Barth. A Davis-Putnam Enumeration Algorithm for Linear Pseudo-Boolean Optimization. Technical Report MPI-I-95-2-003, Max Plank Institute for Computer Science, 1995.
- [Bar95b] P. Barth. *Logic-Based 0-1 Constraint Programming*. Kluwer Academic Publishers, 1995.

- [BB03] O. Bailleux and Y. Boufkhad. Efficient CNF Encoding of Boolean Cardinality Constraints. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming, CP 2003*, volume 2833 of *LNCS*, pages 108–122, September 2003.
- [BBR06] O. Bailleux, Y. Boufkhad, and O. Roussel. A Translation of Pseudo Boolean Constraints to SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:191–200, 2006.
- [BBR09] O. Bailleux, Y. Boufkhad, and O. Roussel. New encodings of pseudo-boolean constraints into cnf. In *12th International Conference on Theory and Applications of Satisfiability Testing, SAT '09*, volume 5584 of *Lecture Notes in Artificial Intelligence*, pages 181–194. Springer, 06 2009.
- [BCCN96] E. Balas, S. Ceria, G. Cornuéjols, and N. Natraj. Gomory Cuts Revisited. *Operations Research Letters*, 19:1–9, 1996.
- [BH02] E. Boros and P. L. Hammer. Pseudo-Boolean Optimization. *Discrete Applied Mathematics*, 123(1-3):155–225, 2002.
- [Bix94] R. E. Bixby. Progress in Linear Programming. *ORSA Journal on computing*, 6(1):15–22, 1994.
- [BJ80] E. Balas and R. G. Jeroslow. Strengthening cuts for mixed-integer programs. *European Journal of Operational Research*, 4:224–234, 1980.
- [BM84a] E. Balas and J. B. Mazzola. Nonlinear 0-1 programming: I. Linearization techniques. *Mathematical Programming*, 30(1):1–21, 1984.
- [BM84b] E. Balas and J. B. Mazzola. Nonlinear 0-1 programming: II. Dominance relations and algorithms. *Mathematical Programming*, 30(1):22–45, 1984.
- [BN15] N. Bjørner and N. Narodytska. Maximum satisfiability using cores and correction sets. In *International Joint Conference on Artificial Intelligence*, pages 246–252. AAAI Press, 2015.
- [BS94] B. Benhamou and L. Saïs. Two proof procedures of cardinality based language in propositional calculus. In E. W. M. e. K. W. W. e. P. Enjalbert, editor, *Proceedings 11th Int. Symposium on Theoretical Aspects of Computer Science (STACS-94)*, volume 775, pages 71–82, Caen, France, February 1994. Springer Verlag.
- [CCD95] S. Ceria, G. Cornuéjols, and M. Dawande. Combining and Strengthening Gomory Cuts. In E. Balas and J. C. (eds.), editors, *Proceedings of the 4th International IPCO Conference on Integer Programming and Combinatorial Optimization*, volume 920 of *Lecture Notes in Computer Science*, pages 438–451. Springer-Verlag, London, UK, May 1995.
- [Chv73] V. Chvátal. Edmonds polytopes and a hierarchy of combinatorial problems. *Discrete Mathematics*, 4:305–337, 1973.
- [CK03] D. Chai and A. Kuehlmann. A Fast Pseudo-Boolean Constraint Solver. In *Proceedings of the Design Automation Conference*, pages 830–835, June 2003.
- [CK05] D. Chai and A. Kuehlmann. A Fast Pseudo-Boolean Constraint Solver. *IEEE Transactions on Computer-Aided Design of Integrated*

- Circuits and Systems*, 24(3):305–317, 2005.
- [CM95] O. Coudert and J. C. Madre. New Ideas for Solving Covering Problems. In *DAC'95: Proceedings of the 32nd ACM/IEEE conference on Design automation*, pages 641–646, New York, NY, USA, June 1995. ACM.
- [Cou93] O. Coudert. Two-Level Logic Minimization, An Overview. *Integration, The VLSI Journal*, 17(2):677–691, October 1993.
- [Cou96] O. Coudert. On Solving Covering Problems. In *Proceedings of the Design Automation Conference*, pages 197–202, June 1996.
- [Dan57] G. B. Dantzig. Discrete-Variable Extremum Problems. *Operations Research*, 5:266–277, 1957.
- [dK86] J. de Kleer. An Assumption-Based TMS. *Artificial Intelligence*, 28(2):127–162, March 1986.
- [DP60] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the Association for Computing Machinery*, 7:201–215, July 1960.
- [ES06] N. Eén and N. Sörensson. Translating Pseudo-Boolean Constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:2–26, 2006.
- [For60] R. Fortet. Application de l'algèbre de Boole en recherche opérationnelle. *Revue Française de Recherche Opérationnelle*, 4:17–26, 1960.
- [Gom58] R. E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64:275–278, 1958.
- [Gom60] R. E. Gomory. An Algorithm for the Mixed-Integer Problem. Technical Report RM-2597, Rand Corporation, 1960.
- [Gom63] R. E. Gomory. An algorithm for integer solutions to linear programs. In *Recent Advances in Mathematical Programming*, pages 269–302. McGraw-Hill, New York, 1963.
- [Hak85] A. Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.
- [Hoo00] J. Hooker. *Logic-Based Methods for Optimization*. Jon Wiley & Sons, 2000.
- [HR68] P. L. Hammer and S. Rudeanu. *Boolean Methods in Operations Research and Related Areas*. Springer Verlag, New York, 1968.
- [HS89] P. L. Hammer and B. Simeone. *Combinatorial Optimization*, volume 1403 of *Lecture Notes in Mathematics*, chapter Quadratic functions of binary variables, pages 1–56. Springer, 1989.
- [Kar72] R. M. Karp. Reducibility Among Combinatorial Problems. In R. E. Miller and J. W. T. (eds.), editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.
- [LD97] S. Liao and S. Devadas. Solving Covering Problems Using LPR-Based Lower Bounds. In *Proceedings of the Design Automation Conference*, pages 117–120, June 1997.
- [LGS⁺07] M. Lukaszewycz, M. Glaß, T. Streichert, C. Haubelt, and J. Teich. Solving Multiobjective Pseudo-Boolean Problems. In *Proceedings of Tenth International Conference on Theory and Applications of*

- Satisfiability Testing*, volume 4501, pages 56–69. Springer Berlin / Heidelberg, May 2007.
- [LL02] A. N. Letchford and A. Lodi. Strengthening Chvatal-Gomory cuts and Gomory fractional cuts. *Operations Research Letters*, 30(2):74–82, 2002.
- [LMS05] I. Lynce and J. P. Marques-Silva. Efficient Data Structures for Backtrack Search SAT Solvers. *Annals of Mathematics and Artificial Intelligence*, 43:137–152, 2005.
- [LW66] E. L. Lawler and D. E. Wood. Branch-and-Bound Methods: A Survey. *Operations Research*, 14(4):699–719, 1966.
- [MHL⁺13] A. Morgado, F. Heras, M. Liffiton, J. Planes, and J. Marques-Silva. Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints*, 18(4):478–534, 2013.
- [MMS04] V. Manquinho and J. P. Marques-Silva. Satisfiability-Based Algorithms for Boolean Optimization. *Annals of Mathematics and Artificial Intelligence*, 40(3-4):353–372, March 2004.
- [MMS05] V. Manquinho and J. P. Marques-Silva. Effective Lower Bounding Techniques for Pseudo-Boolean Optimization. In *Proceedings of the Design and Test in Europe Conference*, pages 660–665, March 2005.
- [MMS06] V. Manquinho and J. P. Marques-Silva. On Using Cutting Planes in Pseudo-Boolean Optimization. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:209–219, 2006. Special Issue on SAT 2005 competition and evaluations.
- [MMSP09] V. M. Manquinho, J. P. Marques-Silva, and J. Planes. Algorithms for Weighted Boolean Optimization. In *Proceedings of the Twelfth International Conference on Theory and Applications of Satisfiability Testing*, pages 495–508. Springer, 2009.
- [MMZ⁺01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the Design Automation Conference*, pages 530–535, June 2001.
- [MP07] J. Marques-Silva and J. Planes. On Using Unsatisfiability for Solving Maximum Satisfiability. *CoRR*, 2007.
- [MR07] V. Manquinho and O. Roussel. Pseudo-Boolean Evaluation 2007. <http://www.cril.univ-artois.fr/PB07>, 2007.
- [MSS96] J. P. Marques-Silva and K. A. Sakallah. GRASP: A New Search Algorithm for Satisfiability. In *Proceedings of the International Conference on Computer Aided Design*, pages 220–227, November 1996.
- [NW88] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, 1988.
- [Sin05] C. Sinz. Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming, CP 2005*, October 2005.
- [SS77] R. M. Stallman and G. J. Sussman. Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis. *Artificial Intelligence*, 9:135–196, October 1977.
- [SS05] H. Sheini and K. Sakallah. Pueblo: A Modern Pseudo-Boolean SAT

- Solver. In *Proceedings of the Design and Test in Europe Conference*, pages 684–685, March 2005.
- [SS06] H. Sheini and K. Sakallah. Pueblo: A Hybrid Pseudo-Boolean SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:157–181, 2006. Special Issue on SAT 2005 competition and evaluations.
- [SV93] T. Schiex and G. Verfaillie. Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. In *Proceedings of the International Conference on Tools with Artificial Intelligence*, pages 48–55, November 1993.
- [VKBSV97] T. Villa, T. Kam, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Explicit and Implicit Algorithms for Binate Covering Problems. *IEEE Transactions on Computer Aided Design*, 16(7):677–691, July 1997.
- [War98] J. P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, 68(2):63–69, 1998.
- [Wil76] H. P. Williams. Fourier-Motzkin Elimination Extension to Integer Programming Problems. *Journal of Combinatorial Theory*, 21:118–123, 1976.
- [Zha97] H. Zhang. SATO: An Efficient Propositional Prover. In *Proceedings of the International Conference on Automated Deduction*, pages 272–275, July 1997.
- [ZM88] R. Zabih and D. A. McAllester. A Rearrangement Search Strategy for Determining Propositional Satisfiability. In *Proceedings of the National Conference on Artificial Intelligence*, pages 155–160, July 1988.
- [ZMMM01] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *Proceedings of the International Conference on Computer Aided Design*, pages 279–285, November 2001.

This page intentionally left blank

Chapter 29

Theory of Quantified Boolean Formulas

Hans Kleine Büning and Uwe Bubeck

29.1. Introduction

Quantified Boolean formulas are a natural extension of propositional formulas by allowing universal (\forall) and existential (\exists) quantifiers. Roughly speaking, a formula $\exists x\phi$ is true if there is a truth assignment to the variable x , such that ϕ is true for this truth value. Analogously, $(\forall x\phi)$ is true if ϕ is true for all truth values. The use of quantifiers results in a greater expressive power, but also in more complex problems. For instance, the satisfiability of a propositional formula ϕ over the variables x_1, \dots, x_n can be expressed by $\exists x_1 \dots \exists x_n \phi$ being true, and the question whether two propositional formulas α and β are logically equivalent can be represented as $\forall x_1 \dots \forall x_n (\alpha \leftrightarrow \beta)$ being true.

This chapter is devoted to foundations and the theory of quantified Boolean formulas and organized as follows: First, we introduce the syntax and the semantics of quantified Boolean formulas including various normal forms. Then we present some complexity results mainly for the satisfiability and the equivalence problem for the class of quantified Boolean formulas and some subclasses. Thereafter, we discuss the expressive power, and based on a more functional view of the valuation of formulas, we show some results on satisfiability and equivalence models. Then we introduce Q-resolution as an extension of the resolution calculus for propositional formulas. The last section is devoted to two special subclasses of quantified Boolean formulas.

29.2. Syntax and Semantics

We will now introduce quantified Boolean formulas. Formulas will be denoted by upper or lower Greek letters, and variables by lower case Latin letters. Besides the universal quantifier \forall and the existential quantifier \exists , the inductive definition makes use of the logical connectives \wedge (and), \vee (or), and the negation \neg . Additionally, for technical reasons, quantified Boolean formulas may contain the Boolean constants true (1) and false (0).

Definition 29.2.1. *Quantified Boolean formulas (QBF^{*})*

The set of *quantified Boolean formulas* is inductively defined as follows:

1. Propositional formulas and the Boolean constants 1 (true) and 0 (false) are quantified Boolean formulas.
2. If Φ is a quantified Boolean formula, then $\exists x\Phi$ and $\forall x\Phi$ are quantified Boolean formulas.
3. If Φ_1 and Φ_2 are quantified Boolean formulas, then $\neg\Phi_1$, $\Phi_1 \vee \Phi_2$ and $\Phi_1 \wedge \Phi_2$ are quantified Boolean formulas.
4. Only formulas given by (1) to (3) are quantified Boolean formulas.

A literal is a variable or a negated variable. We say, a literal L is positive if L is a variable, and a negative literal otherwise. The set of variables in a formula Φ is denoted as $var(\Phi)$. For $\forall x\phi$ (resp. $\exists x\phi$), the occurrence of x in $\forall x$ (resp. $\exists x$) is called a quantified occurrence, where ϕ is the scope of the quantified variable x . An occurrence of the variable x is bound if the occurrence is in the scope of $\forall x$ or $\exists x$. All the non-quantified occurrences of a variable, which are not in the scope of a quantification, are free occurrences. A variable is called *free* (resp. *bound*) in a formula Φ if there is a free (resp. bound) occurrence in Φ . A formula is termed *closed* if and only if the formula contains no free variables.

In the formula $\Phi = (\forall x(x \vee y)) \wedge (x \vee z)$, the first occurrence of x is a quantified occurrence, the second one is bound, and the third one is a free occurrence. The variable x is free and bound in the formula Φ . The variables y and z are free, but not bound. The scope of the variable x in the quantification $\forall x$ is the formula $(x \vee y)$.

The semantics is defined as an extension of the semantics of propositional formulas. Besides the truth assignment to the free variables, the evaluation of a formula is based on the structure of the formula. In addition to propositional formulas, we have to take into account the cases $\exists x$ and $\forall y$. The idea is that in case of a universal quantification, the formula $\forall x\phi$ is true if the formula ϕ is true for all truth assignments to x . An existential quantification $\exists x\phi$ should be true if there is some truth assignment to x , such that ϕ is true.

Definition 29.2.2. *Evaluation*

Let Φ be a quantified Boolean formula with free variables z_1, \dots, z_n . Then a truth assignment \mathfrak{S} is a mapping $\mathfrak{S}: \{z_1, \dots, z_n\} \rightarrow \{0, 1\}$ which satisfies the following conditions:

$$\begin{aligned}
 \Phi = z_i : & \quad \mathfrak{S}(\Phi) = \mathfrak{S}(z_i) \text{ for } 1 \leq i \leq n \\
 \Phi = 0 : & \quad \mathfrak{S}(\Phi) = 0 \\
 \Phi = 1 : & \quad \mathfrak{S}(\Phi) = 1 \\
 \Phi = \neg\Phi' : & \quad \mathfrak{S}(\neg\Phi') = 1 \iff \mathfrak{S}(\Phi') = 0 \\
 \Phi = \Phi_1 \vee \Phi_2 : & \quad \mathfrak{S}(\Phi_1 \vee \Phi_2) = 1 \iff \mathfrak{S}(\Phi_1) = 1 \text{ or } \mathfrak{S}(\Phi_2) = 1 \\
 \Phi = \Phi_1 \wedge \Phi_2 : & \quad \mathfrak{S}(\Phi_1 \wedge \Phi_2) = 1 \iff \mathfrak{S}(\Phi_1) = \mathfrak{S}(\Phi_2) = 1 \\
 \Phi = \exists y\Phi' : & \quad \mathfrak{S}(\exists y\Phi') = 1 \iff \mathfrak{S}(\Phi'[y/0]) = 1 \text{ or } \mathfrak{S}(\Phi'[y/1]) = 1 \\
 \Phi = \forall x\Phi' : & \quad \mathfrak{S}(\forall x\Phi') = 1 \iff \mathfrak{S}(\Phi'[x/0]) = \mathfrak{S}(\Phi'[x/1]) = 1.
 \end{aligned}$$

$\Phi'[z_1/a_1, \dots, z_n/a_n]$ denotes the simultaneous substitution of free occurrences of z_i by a_i in Φ' .

Example: Given is the formula $\Phi = \forall x \exists y ((x \vee y \vee z) \wedge (x \vee \neg y))$ and the truth assignment $\mathfrak{S}(z) = 1$ to the free variable z .

$$\mathfrak{S}(\Phi) = 1$$

$$\iff \mathfrak{S}(\exists y(\phi_1 \wedge \phi_2)[x/0]) = 1 \text{ and } \mathfrak{S}(\exists y(\phi_1 \wedge \phi_2)[x/1]) = 1$$

$$\iff (\mathfrak{S}((\phi_1 \wedge \phi_2)[x/0, y/0]) = 1 \text{ or } \mathfrak{S}((\phi_1 \wedge \phi_2)[x/0, y/1]) = 1) \text{ and} \\ (\mathfrak{S}((\phi_1 \wedge \phi_2)[x/1, y/0]) = 1 \text{ or } \mathfrak{S}((\phi_1 \wedge \phi_2)[x/1, y/1]) = 1)$$

$$\iff (\mathfrak{S}((0 \vee 0 \vee z) \wedge (0 \vee 1)) = 1 \text{ or } \mathfrak{S}((0 \vee 1 \vee z) \wedge (0 \vee 0)) = 1) \text{ and} \\ (\mathfrak{S}((1 \vee 0 \vee z) \wedge (1 \vee 1)) = 1 \text{ or } \mathfrak{S}((1 \vee 1 \vee z) \wedge (1 \vee 0)) = 1)$$

$$\iff (\mathfrak{S}(z) = 1 \text{ or } \mathfrak{S}(0) = 1) \text{ and } (\mathfrak{S}(1) = 1 \text{ or } \mathfrak{S}(1) = 1)$$

$$\iff \mathfrak{S}(z) = 1$$

The formula is true for the truth assignment $\mathfrak{S}(z) = 1$ and false for the truth assignment $\mathfrak{S}(z) = 0$.

Definition 29.2.3. A formula $\Phi \in QBF^*$ is *satisfiable* if and only if there is a truth assignment \mathfrak{S} with $\mathfrak{S}(\Phi) = 1$. Φ is termed *unsatisfiable* (or *inconsistent*) if and only if Φ is not satisfiable.

If Φ contains no free variables, we also say that Φ is *true* (respectively *false*) if Φ is satisfiable (respectively unsatisfiable).

A formula Φ_2 is a *consequence* of Φ_1 ($\Phi_1 \models \Phi_2$) if and only if for all truth assignments \mathfrak{S} we have $\mathfrak{S}(\Phi_1) = 1 \implies \mathfrak{S}(\Phi_2) = 1$. Φ_1 is *logically equivalent* to Φ_2 ($\Phi_1 \approx \Phi_2$) if and only if $\Phi_1 \models \Phi_2$ and $\Phi_2 \models \Phi_1$.

Two formulas Φ_1 and Φ_2 are called *satisfiability equivalent* (\approx_{sat}) if and only if (Φ_1 is satisfiable $\iff \Phi_2$ is satisfiable).

Since closed formulas do not contain free variables, two closed formulas are logically equivalent if and only if these formulas are satisfiability equivalent. Obviously, any quantified Boolean formula Φ with free variables z_1, \dots, z_n is satisfiable if and only if after binding the free variables by existential quantifiers the formula is true. That is, Φ is satisfiable if and only if $\exists z_1 \dots \exists z_n \Phi$ is true. The syntax of quantified Boolean formulas allows us to construct the following formulas:

$$\Phi_1 = (\forall x \phi_1) \wedge (\forall x \phi_2) \text{ and } \Phi_2 = (\forall x (\neg x)) \wedge (x \vee y).$$

In Φ_1 , two distinct occurrences of the quantifier \forall have the same variable x . In the second formula Φ_2 , the variable x is free and bound. In order to avoid distinct occurrences of quantifiers having the same variable name or variables occurring both free and bound, we can rename the quantified variables. For instance, in Φ_1 , we replace the first subformula $\forall x \phi_1$ with $\forall x' \phi'_1$, where x' is a new variable and ϕ'_1 is the result of substituting x' for all free occurrences of x . Similarly, we perform this on the formula Φ_2 , replacing $\forall x (\neg x)$ with $\forall x' (\neg x')$. It is straightforward to see that such renamings preserve the logical equivalence between the initial and the resulting formula. Moreover, we can delete quantifiers whose variables do not occur in the scope of the quantifier. For instance, the variable y is not in the scope $(x \vee z)$ of $\exists y$ in the formula $\forall x \exists y (x \vee z)$. So we can delete the quantification $\exists y$. The formulas satisfying the following two conditions are called *cleansed* or *polite*:

1. Two distinct occurrences of quantifiers have distinct quantification variables.

2. Let $Q \in \{\forall, \exists\}$ and $Qx\phi$ be a subformula in Φ , then x is free in ϕ and not free in Φ .

Any quantified Boolean formula can be transformed into a cleansed formula in linear time via renaming. Subsequently, we assume that all formulas satisfy the conditions. In case of formulas with a propositional constant 0 and/or 1, the following simplification rules can be used to reduce the formula

$$\neg 1 \approx 0, \neg 0 \approx 1, \Phi \wedge 0 \approx 0, \Phi \vee 0 \approx \Phi, \Phi \wedge 1 \approx \Phi, \Phi \vee 1 \approx 1$$

A formula Φ is in *negation normal form* (NNF) if and only if every negation symbol occurs immediately in front of a variable. For instance, $\forall x(\neg x \vee z)$ is in NNF, whereas $\neg(\exists x(x \vee z))$ and $\exists y(\neg\neg y)$ are not in negation normal form. In addition to De Morgan's law and the negation law, which are sufficient for propositional formulas, we have to switch the quantifiers. Any quantified Boolean formula can be transformed into a formula in NNF by the following rules:

$$\begin{aligned} \neg(\Phi_1 \vee \Phi_2) &\approx \neg\Phi_1 \wedge \neg\Phi_2 \text{ and } \neg(\Phi_1 \wedge \Phi_2) \approx \neg\Phi_1 \vee \neg\Phi_2 \\ \neg\neg\Phi &\approx \Phi \\ \neg(\exists x\phi) &\approx \forall x\neg\phi \\ \neg(\forall x\phi) &\approx \exists x\neg\phi \end{aligned}$$

A quantified Boolean formula Φ is in *prenex (normal) form* if the formula consists of a sequence of quantifiers (the *prefix*) followed by a quantifier-free propositional formula, the so-called *matrix* or kernel. A formula Φ in prenex form can be written as $\Phi = Q_1 z_1 \dots Q_n z_n \phi$, where $Q_i \in \{\forall, \exists\}$, z_i are variables and ϕ is the matrix. We sometimes use the notation $\Phi = Q\phi$, where Q is the prefix of the formula and ϕ is the matrix.

If the matrix of a quantified Boolean formula has a particular structure, the abbreviation BF in QBF is replaced with the abbreviation of that class. For example, QCNF* denotes the class of quantified Boolean formulas in prenex form with matrix in CNF. CNF is the class of propositional formulas in conjunctive normal form. HORN is the class of propositional Horn formulas, that means any clause contains at most one positive literal. k -CNF is the class of propositional formulas consisting of k -clauses. A k -clause is a disjunction of at most k literals. If we restrict ourselves to closed formulas, i.e. formulas without free variables, we omit the asterisk *. For instance, Q3-CNF is the class of closed formulas with matrix in 3-CNF.

The length of a quantified Boolean formula is the number of occurrences of variables. Please notice that this definition includes the variables in the prefix. We write $|\Phi|$ in order to denote the length of Φ .

Every quantified Boolean formula can be transformed into a logically equivalent formula in prenex form. For a formula Φ , we first construct a logically equivalent formula Φ' in negation normal form. In general, $(\forall x\phi_1) \vee (\forall x\phi_2)$ is not logically equivalent to $\forall x(\phi_1 \vee \phi_2)$. But that case cannot occur, because we are dealing with cleansed formulas. That is, distinct occurrences of quantifiers have distinct variable names.

In order to generate formulas in prenex form, we can apply the following rules to a formula Φ' :

$$\begin{aligned} (\forall x\phi) \vee \Phi &\approx \forall x(\phi \vee \Phi) \\ (\forall x\phi) \wedge \Phi &\approx \forall x(\phi \wedge \Phi) \\ (\exists x\phi) \wedge \Phi &\approx \exists x(\phi \wedge \Phi) \\ (\exists x\phi) \vee \Phi &\approx \exists x(\phi \vee \Phi) \end{aligned}$$

More sophisticated prenexing strategies can be borrowed from procedures well-known for first-order logic or can be found specially for QBFs for example in [EST⁺04].

Lemma 29.2.1. *Any quantified Boolean formula Φ can be transformed into a logically equivalent formula which is cleansed and in prenex form. The time required is linear in the length of Φ .*

It is well-known that any propositional formula in NNF can be transformed into a logically equivalent formula in CNF by means of the distributive laws $(\alpha \vee \beta) \wedge \sigma \approx (\alpha \wedge \sigma) \vee (\beta \wedge \sigma)$ and $(\alpha \wedge \beta) \vee \sigma \approx (\alpha \vee \sigma) \wedge (\beta \vee \sigma)$. We can apply these rules to the propositional matrix of a quantified Boolean formula in prenex form and obtain a formula in QCNF*. But the problem is the possibly exponential growth of the formula. In order to avoid this critical process, we make use of the well-known procedure introduced in [Tse70].

For propositional formulas in NNF, we replace subformulas of the form $\alpha \vee (\beta \wedge \gamma)$ with $((\alpha \vee \neg x) \wedge (\beta \vee x) \wedge (\gamma \vee x))$, where x is a new variable. Finally, we obtain a formula in CNF. Both formulas are satisfiability equivalent, but in general not logically equivalent. If we bind the new variable x by an existential quantifier, we have

$$\alpha \vee (\beta \vee \gamma) \approx \exists x((\alpha \vee \neg x) \wedge (\beta \vee x) \wedge (\gamma \vee x)).$$

Let $\Phi = Q\phi$ be a formula in prenex form, x_1, \dots, x_n the new variables introduced by the above procedure, and ϕ' the conjunction of the derived clauses. Then we have

$$\Phi \approx Q\exists x_1 \dots \exists x_n \phi' \text{ in QCNF*}.$$

Since three new literals are added for every subformula of the form $\alpha \vee (\beta \wedge \gamma)$ in Φ and the number of such subformulas is bounded by the length of Φ , the length of the resulting CNF formula is linear in the length of Φ .

Similar to the propositional case, we can transfer arbitrary QCNF* formulas into logically equivalent Q3-CNF* formulas in linear time as follows. Let $\Phi = Q\phi$ be a formula in QCNF*. We replace any clause $\varphi = (L_1 \vee \dots \vee L_n)$ containing more than three literals with the sequence of clauses $\varphi' = (L_1 \vee L_2 \vee x_1), (\neg x_1 \vee L_2 \vee x_2), (\neg x_2 \vee L_3 \vee x_3), \dots, (\neg x_{n-2} \vee L_{n-1} \vee L_n)$ for new variables x_1, \dots, x_{n-2} . Binding the newly introduced variables existentially, we obtain not only satisfiability equivalence, but also logical equivalence $\varphi \approx \exists x_1 \dots \exists x_{n-2} \varphi'$.

Let ϕ' be the result of replacing all clauses of length greater than 3 with the conjunction of 3-clauses, and let x_1, \dots, x_m be the new variables. Then we have

$$\Phi \approx Q\exists x_1 \dots \exists x_m \phi' \text{ in Q3-CNF*}.$$

Besides Q3-CNF*, for every fixed $k \geq 3$, we can generate a logically equivalent formula in Q k -CNF* by adapting the substitution to k -clauses. But for Q2-CNF*, no polynomial-time procedure is known, and assuming that PSPACE does not equal P, such a transformation cannot exist. We summarize the previous remarks on transformations in the following lemma.

Lemma 29.2.2. *For $k \geq 3$, any QBF* formula Φ can be transformed into a logically equivalent QCNF* and Q k -CNF*. The time is linear in the length of Φ .*

Besides the above mentioned simplifications and transformations, a formula in QCNF* can be reduced according to the following observations. Let Φ be a formula in QCNF*:

1. If Φ is not a tautology, all tautological clauses can be deleted preserving the logical equivalence.
2. If Φ contains a non-tautological clause with only universal variables, then the formula is unsatisfiable (false).
3. *Universal Reduction / ForAll Reduction:*

Suppose Φ contains a non-tautological clause $(\varphi \vee x)$, where x is a universally quantified variable and φ is a subclause. If there is no existentially quantified variable in φ , or if every existentially quantified variable in φ occurs in the prefix before the variable x , we can delete the variable x while preserving the logical equivalence.

An example is the formula $\exists y \forall x \exists y_1 ((\neg y \vee \neg x) \wedge (y \vee \neg x \vee y_1) \wedge (z \vee x))$. In the first clause, $\neg x$ can be deleted, because y occurs in the prefix before x . In the second clause, x is followed by y_1 . The clause remains unchanged. And in the last clause, we have the free variable z but no existentially quantified variable. We can remove x . The result is the formula $\exists y \forall x \exists y_1 ((\neg y) \wedge (y \vee \neg x \vee y_1) \wedge z)$.

4. *Unit Propagation:*

If Φ contains a clause $C = y^\epsilon$ with a single existential literal y^ϵ ($y^0 := \neg y$ and $y^1 := y$), then logical equivalence is preserved when we assign $y = \epsilon$ by replacing all occurrences of y^ϵ in ϕ with 1 and all occurrences of $y^{1-\epsilon}$ with 0. In the formula from above, $\neg y$ is an existential unit which can be propagated. We obtain the formula $\forall x \exists y_1 ((\neg x \vee y_1) \wedge z)$.

5. *Pure Literal Detection:*

If Φ contains a quantified variable v which occurs only positively or only negatively in ϕ , then logical equivalence is preserved in the following cases:

- (a) if v is existentially quantified and all clauses containing v are deleted from Φ .
- (b) or if v is universally quantified and the occurrences of v are removed from all clauses of ϕ .

In the previous example, y_1 occurs only positively, so we can delete the first clause. We can conclude that the formula is equivalent to just z .

29.3. Complexity Results

The satisfiability problem for quantified Boolean formulas, often denoted as QSAT, is the problem of deciding whether or not a quantified Boolean formula is satisfiable. It was shown in [MS73] that for quantified Boolean formulas without free variables, QSAT is solvable in polynomial space and complete for PSPACE. By the previous remarks on transformations of QBF* into QCNF* and Q3-CNF*, we see that the satisfiability problem for Q3-CNF* is PSPACE-complete, too. Since $\text{PSPACE} = \text{NPSPACE}$ [Sav70], that means problems solvable non-deterministically in polynomial space are also solvable deterministically in polynomial space, we immediately obtain the PSPACE-completeness of the complement of QSAT, the set of unsatisfiable formulas. According to this observation, we can conclude that the consequence and the equivalence problems are also PSPACE-complete. The consequence problem (respectively equivalence problem) is to decide whether for arbitrarily given formulas Φ_1 and Φ_2 we have $\Phi_1 \models \Phi_2$ (respectively $\Phi_1 \approx \Phi_2$).

Theorem 29.3.1. *The following problems are PSPACE-complete:*

1. *Satisfiability problem for QBF* and Q3-CNF**
2. *Consequence problem for QBF*.*
3. *Equivalence problem for QBF*.*

In contrast to propositional formulas, for which the satisfiability problem for formulas in disjunctive normal form (DNF) is solvable in polynomial time, the satisfiability problem for QDNF remains PSPACE-complete. That follows directly from the fact that the unsatisfiability problem for QCNFs remains PSPACE-complete, and a negation of formulas in QCNF can be transferred into a formula in QDNF by moving the negation symbols inside.

From an application perspective, it is interesting to notice that various verification problems are also PSPACE-complete, e.g. propositional linear temporal logic (LTL) satisfiability [SC85] or symbolic reachability in sequential circuits [Sav70]. Thus it seems quite natural to solve such problems by encoding them as quantified Boolean formulas.

The satisfiability problem for Q2-CNF* and for QHORN* is solvable in polynomial time, which will be discussed in Section 29.6. In order to reduce the worst-case complexity of the satisfiability problem, mainly two approaches have been considered. One is to restrict the matrix to certain subclasses of propositional formulas, and the second is to consider bounded alternations of quantifiers. Next, we introduce the prefix type of formulas in prenex form. Roughly speaking, Σ_n (respectively Π_n) is the set of formulas with an outermost existential (respectively universal) quantifier and n alternations.

Definition 29.3.1. *Prefix Type of Quantified Boolean Formulas*

Every propositional formula has the prefix type $\Sigma_0 = \Pi_0$ (empty prefix). Let Φ be a quantified Boolean formula with prefix type Σ_n (respectively Π_n), then the formula $\forall x_1 \cdots \forall x_n \Phi$ (respectively $\exists y_1 \cdots \exists y_n \Phi$) is of type Π_{n+1} (respectively Σ_{n+1}).

For instance, the formula $\Phi = \forall x \exists y \forall z \phi$ with matrix ϕ is in Π_3 , because ϕ is in Σ_0 , $\forall z \phi \in \Pi_1$, and $\exists y \forall z \phi \in \Sigma_2$. Please note that any closed quantified Boolean formula containing at least one variable is of prefix type Σ_k or Π_k with $k \geq 1$.

There is a strong connection to the polynomial-time hierarchy, introduced in [MS72]. The polynomial-time hierarchy is defined as follows ($k \geq 0$):

$$\Delta_0^P := \Sigma_0^P := \Pi_0^P := P$$

$$\Sigma_{k+1}^P := NP^{\Sigma_k^P}, \quad \Pi_{k+1}^P := co\Sigma_{k+1}^P, \quad \Delta_{k+1}^P := P^{\Sigma_k^P}$$

Δ_{k+1}^P (respectively Σ_{k+1}^P) is the class of all problems which can be decided deterministically (respectively non-deterministically) in polynomial time with the help of an oracle for a problem in Σ_k^P . An oracle is a subroutine which solves a problem in the complexity class Σ_k^P . This subroutine is called in the program, but it contributes only $O(1)$ to the complexity of this program. The class Π_{k+1}^P contains every problem whose complement is in Σ_{k+1}^P . We have $\Sigma_1^P = NP$, $\Pi_1^P = coNP$ and $\Delta_1^P = P$. For $k \geq 1$, we obtain $\Delta_k^P \subseteq \Sigma_k^P \cap \Pi_k^P$. The following theorem has been proved in [Sto76, Wra76].

Theorem 29.3.2. *For $k \geq 1$, the satisfiability problem for quantified Boolean formulas with prefix type Σ_k is Σ_k^P -complete, and for formulas with prefix type Π_k , it is Π_k^P -complete.*

Another approach to characterize classes of quantified Boolean formulas with respect to the complexity of the satisfiability problem is the so-called dichotomy theorem [Sch78]. Let C be a set of Boolean functions $\{f_1, \dots, f_n\}$, where the n_i -ary functions f_i are often called *constraints*. For $1 \leq i \leq n$ let \mathbf{x}_i be a n_i -tuple of not necessarily distinct variables. Then $f_i(\mathbf{x}_i)$ is called a constraint application, and a finite conjunction of constraint applications is denoted as $F(y_1, \dots, y_t)$, where y_1, \dots, y_t are the occurring variables.

We say, the expression $F(y_1, \dots, y_t)$ is satisfiable if and only if there is a truth assignment to the variables y_i , such that the expression is true. Obviously, the complexity of the satisfiability problem for such classes of expressions essentially depends on the set of Boolean functions C . A *quantified constraint expression* has the form

$$Q_1 y_1 \dots Q_n y_n F(y_1, \dots, y_n), \text{ where } Q_i \in \{\forall, \exists\}.$$

Let C be a finite set of constraints. QC is the set of quantified expressions $Q_1 y_1 \dots Q_n y_n F(y_1, \dots, y_n)$, where $Q_i \in \{\forall, \exists\}$ and $F(y_1, \dots, y_n)$ is a conjunction of constraint applications from C . If the constraint applications can include nullary functions (the propositional constants 0 and 1), the class is denoted as $Q_c(C)$. Please notice that QC as well as $Q_c(C)$ do not contain formulas with free variables.

We next classify constraints according to whether the constraints (Boolean functions) can be represented as propositional formulas of a special form.

Definition 29.3.2. Let f be a constraint.

1. f is *Horn* if and only if f is equivalent to a Horn formula.
2. f is *anti-Horn* if and only if f is equivalent to a conjunction of clauses which contain at most one negative literal.

3. f is *bijunctive* if and only if f is equivalent to a 2-CNF formula.
4. f is *affine* if and only if f is equivalent to a XOR-CNF formula, i.e. a conjunction of clauses of the form $(x_1 \oplus \dots \oplus x_n = 1)$ or $(x_1 \oplus \dots \oplus x_n = 0)$, where \oplus denotes addition modulo 2.

A finite set of constraints C is called Horn (anti-Horn, bijunctive, affine, respectively), if every constraint in C is Horn (anti-Horn, bijunctive, affine, respectively). The satisfiability problem for quantified constraint applications is denoted as QSAT(C), and in case of applications with constants as QSAT $_c$ (C).

Theorem 29.3.3. Dichotomy Theorem [Sch78, Dal97, CKS01]

Let C be a finite set of constraints. If C is Horn, anti-Horn, bijunctive, or affine, then QSAT(C) and QSAT $_c$ (C) are in P, otherwise QSAT(C) and QSAT $_c$ (C) are PSPACE-complete.

The Dichotomy Theorem immediately implies that the satisfiability problem for Q2-CNF is solvable in polynomial time. For instance, let $C_2 = \{f_1, f_2, f_3\}$ be the set of Boolean functions, where the functions are given as 2-clauses

$$f_1(x, y) = x \vee y, f_2(x, y) = (\neg x \vee y), \text{ and } f_3(x, y) = (\neg x \vee \neg y).$$

Every conjunction of constraint applications coincides to a formula in 2-CNF and vice versa. For instance,

$$f_1(x_1, x_3) \wedge f_1(x_2, x_2) \wedge f_2(x_3, x_2) \approx (x_1 \vee x_3) \wedge (x_2 \vee x_2) \wedge (\neg x_3 \vee x_2).$$

That shows that QSAT(C_2) is in P if and only if the satisfiability problem for Q2-CNF is polynomial-time solvable.

In QHORN, clauses may have an arbitrary length and can therefore not directly be represented by a finite set of constraints. But every formula in QHORN can be transformed into an equivalent Q3-HORN formula. To the possible types of 3-clauses, we can associate a finite set of constraints \mathcal{H} , which is Horn. For instance, for 3-clauses with one positive literal, we define $f(x_1, x_2, x_3) = (x_1 \vee \neg x_2 \vee \neg x_3)$. Since the satisfiability problem QSAT(\mathcal{H}) is in P, we obtain the polynomial-time solvability of the satisfiability problem for QHORN. In [Sch78], the dichotomy theorem has been shown for non-quantified expressions. For QSAT(C) and QSAT $_c$ (C), the theorem is presented, but no complete proof is given. More about the dichotomy theorem, including the proof of the theorem, can be found, among others, in [Dal97, CKS01, FK06]. Results for formulas with a fixed prefix type have been shown in [Hem04].

An example of a subclass of quantified Boolean formulas whose satisfiability problem is not covered by the Dichotomy Theorem is the set of formulas in which the matrix is a conjunction of prime implicates. A *prime implicate* of a CNF formula ϕ is a clause π such that $\phi \models \pi$, and for any proper subclause $\pi' \subset \pi$, we have $\phi \not\models \pi'$. A *prime implicate formula* is a CNF formula ϕ where every clause is a prime implicate and every prime implicate is a clause in ϕ . The set of prime implicate formulas is denoted as PI. The class PI can be decided in polynomial time, because a CNF formula ϕ is in PI if and only if the following conditions hold:

1. no clause in ϕ properly contains another clause in ϕ .
2. for every resolvent α from any two clauses in ϕ , there is a clause $\pi \in \phi$ with $\pi \subseteq \alpha$.

And since every unsatisfiable PI formula contains the empty clause, the satisfiability problem for PI is polynomial-time solvable.

The quantified extension of prime implicates formulas are formulas of the form $Q_1x_1 \dots Q_nx_n\phi$, where $Q_i \in \{\forall, \exists\}$ and ϕ is a prime implicates formula over the variables x_1, \dots, x_n . The set of these formulas is denoted as $Q(\text{PI})$. The satisfiability problem for $Q(\text{PI})$ is PSPACE-complete. That has been shown in [CMLBLM06] by a reduction from the satisfiability problem for QCNF.

It is well-known that every propositional formula can be represented as a graph. We now assume that graphs are DAGs (directed acyclic graphs). The leaves are labeled with literals x or $\neg x$ or with the propositional constants 1 or 0. The internal nodes are labeled with \wedge or \vee . Let C be a node, then $\text{var}(C)$ denotes the set of all variables that label the descendants of node C . If C is the root, $\text{var}(C)$ is the set of all variables occurring in the graph. For sake of simplicity, a graph is called a formula. Depending on the structure and on various constraints, special graphs and corresponding classes of propositional formulas have been investigated. Moreover, the graph representation allows us, for instance, to reason about the satisfiability problem and related problems for the quantified version of ordered binary decision diagrams (OBDDs) and free binary decision diagrams (FBDDs). Most proofs of the PSPACE-completeness of the satisfiability problem for the quantified classes are based on the NP-completeness of the satisfiability problem for the matrices. As an example, we present two classes of formulas:

Let DNNF be the class of decomposable formulas, which means the conjuncts of every \wedge -node C do not share variables. That is, if C_1, \dots, C_n are the successors of an \wedge -node C , then $\text{var}(C_i) \cap \text{var}(C_j)$ is empty for $i \neq j$.

The class d-DNNF is the set of formulas in DNNF with the additional condition of determinism. That is, for every \vee -node C , if C_i and C_j are successors of C , then $C_i \wedge C_j$ is inconsistent for $i \neq j$.

Let QDDNF (Qd-DNNF, QFBDD, QOBDD $_{<}$ respectively) be the set of quantified formulas in prenex form with matrix in DNNF (d-DNNF, FBDD, OBDD $_{<}$). Then the following result has been shown.

Theorem 29.3.4. [CMLBLM06]

The satisfiability problems for QDNNF, Qd-DNNF, QFBDD, and QOBDD $_{<}$ are PSPACE-complete.

29.4. Models and Expressive power

Besides the natural description of quantified propositions, QBF is a tool for short representations of propositional formulas and Boolean functions. Any quantified Boolean formula with free variables z_1, \dots, z_n is logically equivalent to a propositional formula and a Boolean function over these variables. In propositional logic, auxiliary variables are often used to reduce the length of a formula. For

example, the propositional formula $\phi = \bigwedge_{1 \leq i, j \leq n} (a_i \vee b_j)$ has length $2 \cdot n^2$. For a new variable y , the formula

$$\Phi = \exists y \left(\bigwedge_{1 \leq i \leq n} (a_i \vee y) \wedge \bigwedge_{1 \leq j \leq n} (\neg y \vee b_j) \right)$$

is logically equivalent to ϕ and of length $4 \cdot n$. In the best case, an exponential reduction can be obtained. A technique to reduce the length by means of a more complex prefix is based on the following idea. We use the universal variables to exclude some consequences. In the example above, suppose we want to exclude the clause $(a_1 \vee b_3)$ and the clause $(a_2 \vee b_2)$. Then we define $\Phi' = \forall x_1 \forall x_2 \exists y ((a_1 \vee x_1 \vee y) \wedge (a_2 \vee x_2 \vee y) \wedge (a_3 \vee y) \wedge (\neg y \vee b_1) \wedge (\neg x_2 \vee \neg y \vee b_2) \wedge (\neg x_1 \vee \neg y \vee b_3))$.

Since the clause $(a_1 \vee x_1 \vee y)$ contains x_1 , and $\neg x_1$ is in the clause $(\neg x_1 \vee \neg y \vee b_3)$, $(a_1 \vee b_3)$ is not a consequence of Φ' . Similarly, the second and the fifth clause contain the complementary pair of universal literals x_2 and $\neg x_2$. Therefore, $(a_2 \vee b_2)$ is not an implication. All the other consequences remain valid.

Any propositional formula and quantified Boolean formula is a representation of a Boolean function. The representation of Boolean functions and the compression of formulas have their limits. Not every n -ary Boolean function f can be described by a formula $\Phi \in \text{QBF}^*$ which is polynomial in the length of n . That is, there is no polynomial p , such that for every Boolean function $f : \{0, 1\}^n \rightarrow \{0, 1\}$ there exists a quantified Boolean formula $\Phi \in \text{QBF}^*$ with free variables z_1, \dots, z_n which satisfies

$$f \approx \Phi \text{ and } |\Phi| \leq p(n).$$

That can be shown straightforward by a counting argument, because the number of n -ary Boolean functions is 2^{2^n} , and the number of QBF^* formulas of length $p(n)$ is bound by $2^{q(n)}$ for some polynomial q .

In general, it is not necessarily a good idea to translate a quantified Boolean formula into a logically equivalent propositional formula. We will return to this approach later in the section on quantified Horn formulas.

A more functional view of the valuation of a quantified Boolean formula is based on the role of the existential variables. The value of an existential variable depends upon the current values of the dominating universal variables. For instance, the formula $\Phi = \forall x \exists y ((x \vee y) \wedge (\neg x \vee \neg y))$ is true. If $x = 0$, then we choose $y = 1$, and for $x = 1$ we assign $y = 0$. That can be expressed as a Boolean function $f : \{0, 1\} \rightarrow \{0, 1\}$ with $f(0) = 1$ and $f(1) = 0$, i.e. $f(x) = \neg x$. Replacing the variable y in Φ with the function $f(x)$, we obtain the formula $\Phi' = \forall x ((x \vee f(x)) \wedge (\neg x \vee \neg f(x))) \approx \forall x ((x \vee \neg x) \wedge (\neg x \vee x))$, which is true. In our example, the Boolean function has been represented as a propositional formula.

For sake of simplicity, we subsequently only deal with formulas in prenex form. Let $\Phi = \forall x_1 \exists y_1 \dots \forall x_k \exists y_k \phi$ be a quantified Boolean formula without free variables. Then Φ is true if and only if there exist Boolean functions $f_i : \{0, 1\}^i \rightarrow \{0, 1\}$ for $1 \leq i \leq k$, such that

$$\forall x_1 \dots \forall x_k \phi[y_1/f_1(x_1), \dots, y_k/f_k(x_1, \dots, x_k)] \text{ is true if and only if } \phi[y_1/f_1(x_1), \dots, y_k/f_k(x_1, \dots, x_k)] \text{ is a tautology.}$$

Definition 29.4.1. *Satisfiability Model*

Let $\Phi = \forall x_1 \exists y_1 \dots \forall x_k \exists y_k \phi$ be a quantified Boolean formula without free variables. A sequence of Boolean functions $M = (f_1, \dots, f_k)$ is denoted a *satisfiability model* if and only if $\Phi' = Q\phi[y_1/f_1(x_1), \dots, y_k/f_k(x_1, \dots, x_k)]$ is true.

The definition of satisfiability models can easily be extended to formulas with free variables. In that case, the Boolean functions substituting the free variables are the constants 0 or 1. The syntax of quantified Boolean formulas does not include function symbols for Boolean functions. Such an extension will be discussed later. In a first approach, we assume that the Boolean functions are given as propositional or quantified Boolean formulas. Then after the substitution of the functions, the formulas are again quantified Boolean formulas, but in case of functions given as QBFs, not necessarily in prenex form.

The length of satisfiability models represented as propositional formulas cannot be bound by a polynomial, assuming $\text{PSPACE} \neq \Sigma_2^P$. Suppose there is a polynomial p such that any satisfiable formula Φ has a satisfiability model $M = (f_1, \dots, f_n)$ with $|f_i| \leq p(|\Phi|)$. For a closed formula $\Phi = \forall x_1 \exists y_1 \dots \forall x_n \exists y_n \phi$, we obtain:

Φ is true if and only if $\forall x_1 \dots \forall x_n \phi[y_1/f_1, \dots, y_n/f_n]$ is true if and only if $\phi[y_1/f_1, \dots, y_n/f_n]$ is a tautology.

Moreover, $\Phi[y_1/f_1, \dots, y_n/f_n]$ is of length at most $O(|\Phi| \cdot p(|\Phi|))$. It follows that the truth of Φ can be decided by guessing propositional formulas f_i of length less than or equal to $p(|\Phi|)$ and solving the coNP-complete tautology problem. That implies the PSPACE-complete satisfiability problem for QCNF is in Σ_2^P , in contradiction to the widely believed conjecture $\text{PSPACE} \neq \Sigma_2^P$.

But if we represent the model functions as quantified Boolean formulas, then there is a polynomial p , such that the length of the satisfiability models is bound by the polynomial.

Models are of particular interest, because they provide a precise characterization of the behavior of the existentially quantified variables. Knowing (parts of) the model can be helpful when solving a quantified Boolean formula, because that process requires finding satisfying truth assignments to the existentials for different values of the universal variables. For example, a solver can use models as a formalism to represent information about the existentials which it has learned while traversing the search space. A popular example of this technique is the QBF solver sKizzo [Ben05], which is based on successively computing the model functions and storing them compactly as binary decision diagrams (BDDs).

Another reason for models being an interesting research topic is that there appears to be a close connection between the structure of QBF formulas and the structure of their models. From an application perspective, it is clear that the information that a given formula has models of a special structure can be very useful for a QBF solver to prune the search space. We now present some of these results on the relationship between formula classes and the structure of satisfiability models.

Definition 29.4.2. Let K be a class of propositional formulas and $X \subseteq \text{QCNF}$. Then $M = (f_1, \dots, f_n)$ is called a *K-model* for $\Phi \in X$ if M is a satisfiability model for Φ and the formula f_i is in K for every $1 \leq i \leq n$.

The satisfiability model problems we are interested in are defined as

K -Model Problem for X :

Instance: A formula $\Phi \in X$.

Query: Does there exist a K -model for Φ ?

K -Model Checking Problem for X :

Instance: A formula $\Phi \in X$ and $M = (f_1, \dots, f_n)$ a sequence of propositional formulas $f_i \in K$.

Query: Is M a K -model for Φ ?

Let K be the set of propositional formulas. Then the K -model problem for QCNF is PSPACE-complete, because the problem is equivalent to the satisfiability problem for QCNF. In contrast, the K -model checking problem for QCNF is in coNP, since after replacing the existential variables with the propositional formulas, we have to decide whether the resulting matrix is a tautology. The coNP-completeness follows easily by a reduction of the tautology problem for propositional formulas. For a more detailed analysis, we introduce the following sets K_0, K_1 , and K_2 of Boolean functions, where $x^0 := \neg x$ and $x^1 := x$.

$$K_0 = \{f \mid f \text{ is a constant } 0 \text{ or } 1\}$$

$$K_1 = \{f \mid \exists i \exists \epsilon \in \{0, 1\} : f(x_1, \dots, x_n) = x_i^\epsilon\} \cup K_0$$

$$K_2 = \{f \mid \exists I \subseteq \{1, \dots, n\} : f(x_1, \dots, x_n) = \bigwedge_{i \in I} x_i\} \cup K_0.$$

For the next theorem, we need the following notation. Let $\Phi = Q(\alpha_1 \wedge \dots \wedge \alpha_n)$ and $\Phi' = Q(\alpha'_1 \wedge \dots \wedge \alpha'_n)$ be two formulas in QCNF*. If α'_i is a subclause of α_i for every i , we write $\Phi' \subseteq_{cl} \Phi$. We say Φ' is a subclause-subformula of Φ .

Theorem 29.4.1. [KBSZ04]

1. Every satisfiable formula $\Phi \in \text{Q2-CNF}$ has a K_1 -satisfiability model.
2. A formula $\Phi \in \text{QCNF}$ has a K_1 -satisfiability model if and only if there is some $\Phi' \subseteq_{cl} \Phi$ with $\Phi' \in \text{Q2-CNF}$ and Φ' being satisfiable.
3. Every satisfiable formula $\Phi \in \text{QHORN}$ has a K_2 -satisfiability model.
4. A formula $\Phi \in \text{QCNF}$ has a K_2 -satisfiability model if and only if there is some $\Phi' \subseteq_{cl} \Phi$ with $\Phi' \in \text{QHORN}$ and Φ' being satisfiable.

Theorem 5 characterizes Q2-CNF and QHORN in terms of satisfiability models. We see that every satisfiable formula in Q2-CNF has a model in which every existential variable depends only on at most one universal variable. In case of formulas in QHORN, conjunctions of variables are sufficient (besides the constants). Similar to propositional Horn formulas, the satisfiability models for quantified Horn formulas are closed under intersection, which can be shown straightforward.

Lemma 29.4.2. Let $M_1 = (f_1, \dots, f_n)$ and $M_2 = (g_1, \dots, g_n)$ be two satisfiability models for the formula $\Phi \in \text{QHORN}$. Then $M_1 \cap M_2 := (f_1 \wedge g_1, \dots, f_n \wedge g_n)$ is a model for Φ .

For further subclasses, more results can be found in [KBSZ07]. Sometimes, the term *policy* or *strategy* is used instead of satisfiability model.

Instead of looking for satisfiability models, we may try to determine Boolean functions, such that after the substitution of the functions for the existential variables, the input formula is logically equivalent to the resulting formula. Take for example the formula

$$\Phi = \forall x \exists y ((x \vee y \vee z_1) \wedge (x \vee \neg y \vee z_2) \wedge (\neg x \vee \neg y \vee z_3)).$$

The formula is logically equivalent to $(z_1 \vee z_2)$. When we substitute the function $f(z_1, z_2, z_3, x) = (z_2 \wedge \neg x)$ for y , we obtain the formula $\forall x((x \vee (z_2 \wedge \neg x) \vee z_1) \wedge (x \vee \neg(z_2 \wedge \neg x) \vee z_2) \wedge (\neg x \vee \neg(z_2 \wedge \neg x) \vee z_3))$, and after simplifying the formula $\forall x((x \vee z_1 \vee z_2) \wedge (x \vee \neg z_2 \vee z_2) \wedge (\neg x \vee \neg z_2 \vee x \vee z_3))$. The last two clauses are tautological clauses, and by universal reduction on the first clause, we see that the formula is logically equivalent to the input formula. Please notice that in addition to the universally quantified variables, the Boolean functions now also depend on the free variables.

Definition 29.4.3. *Equivalence Model*

Let $\Phi = \forall x_1 \exists y_1 \dots \forall x_n \exists y_n \phi$ be a quantified Boolean formula with free variables $\mathbf{z} = z_1, \dots, z_t$. A sequence $M = (f_1, \dots, f_n)$ of Boolean functions $f_i \in K$ is called an *equivalence model* for Φ if and only if

$$\Phi \approx \forall x_1 \dots \forall x_n \phi[y_1/f_1(\mathbf{z}, x_1), \dots, y_n/f_n(\mathbf{z}, x_1, \dots, x_n)].$$

Let K be a set of Boolean functions. If every function f_i is in K , we say that M is a K -equivalence model.

The correspondence between formula classes and model structure is especially interesting for equivalence models. As mentioned before, QBF^* formulas can often represent propositional formulas exponentially shorter, and restrictions on the structure of the equivalence model translate to limits in the expressiveness (i.e. the ability to provide short representations for propositional formulas) of the corresponding class of QBF^* formulas.

The following two kinds of problems have been investigated for various subclasses of Boolean functions and quantified Boolean formulas.

K -Equivalence Model Problem for X :

Instance: A formula $\Phi \in X$.

Query: Does there exist a K -equivalence model for Φ ?

K -Equivalence Model Checking Problem for X :

Instance: A formula $\Phi \in X$ and $M = (f_1, \dots, f_n)$ a sequence of propositional formulas $f_i \in K$.

Query: Is M a K -equivalence model for Φ ?

It has been shown [KBZ05] that every quantified Boolean formula has an equivalence model. For closed formulas, any satisfiability model is an equivalence model, too. In general, equivalence models are more complex than satisfiability models.

Lemma 29.4.3. [KBZ05, BKBZ05]

1. Every $Q2$ -CNF * formula has an equivalence model where the functions can be represented as propositional formulas in $1\text{-CNF} \cup 1\text{-DNF} \cup \{0, 1\}$.
2. Every formula in $QHORN^*$ has an equivalence model where the Boolean functions can be represented as monotone propositional formulas.

For the test of whether a sequence of Boolean functions is an equivalence model, for instance the following propositions are known.

Lemma 29.4.4. [KBZ05]

1. The K_0 -equivalence model checking problem for arbitrary QCNF* formulas is PSPACE-complete.
2. Let K be the set of propositional formulas. Then the K -equivalence model checking problems for Q2-CNF* and QHORN* are coNP-complete.

We conclude the section with some remarks on an extension of quantified Boolean formulas. Suppose we have a formula $\Phi = \forall x_1 \forall x_2 \forall x_3 \exists y_1 \exists y_2 \exists y_3 \phi$, and we want to know whether Φ is true when y_1 only depends on x_1 and x_2 , y_2 on x_2 and x_3 , and y_3 on x_1 and x_3 . That is the question whether there are model functions $f_1(x_1, x_2)$, $f_2(x_2, x_3)$, and $f_3(x_1, x_3)$ such that after the substitution, the formula $\forall x_1 \forall x_2 \forall x_3 \phi[y_1/f_1(x_1, x_2), y_2/f_2(x_2, x_3), y_3/f_3(x_1, x_3)]$ is true. It is not clear how these requirements could directly be represented in a succinct QBF*, even if we allow non-prenex formulas. This kind of question can be expressed in an extension of quantified Boolean formulas, the so-called *Dependency Quantified Boolean Formulas* (DQBF).

The idea is based on partially ordered quantifiers introduced for first-order predicate logic [Hen61]. Instead of a total ordering on the quantifiers, we can directly express in a function-style notation on which universal variables an existential variable depends. For the formula above, we write

$$\forall x_1 \forall x_2 \forall x_3 \exists y_1(x_1, x_2) \exists y_2(x_2, x_3) \exists y_3(x_1, x_3) \phi.$$

These quantifiers are also called branching quantifiers or Henkin quantifiers. More formally, a dependency quantified Boolean formula $\Phi \in \text{DQBF}$ with universal variables x_1, \dots, x_n and existential variables y_1, \dots, y_m is a formula of the form

$$\Phi = \forall x_1 \dots \forall x_n \exists y_1(x_{d_{1,1}}, \dots, x_{d_{1,t_1}}) \dots \exists y_m(x_{d_{m,1}}, \dots, x_{d_{m,t_m}}) \phi$$

where $x_{d_{i,j}}$ points to the j^{th} variable on which y_i depends. The formula Φ is said to be true if and only if there are Boolean functions $f_i(x_{d_{i,1}}, \dots, x_{d_{i,t_i}})$ such that

$$\forall x_1 \dots \forall x_n \phi[y_1/f_1(x_{d_{1,1}}, \dots, x_{d_{1,t_1}}), \dots, y_m/f_m(x_{d_{m,1}}, \dots, x_{d_{m,t_m}})]$$

is true.

It has been shown in [PRA01] that the problem of deciding whether an arbitrarily given $\Phi \in \text{DQBF}$ is true is NEXPTIME-complete. Assuming that NEXPTIME does not equal PSPACE, this means that there are DQBF formulas for which no logically equivalent QBF* of polynomial length can be computed in polynomial space. For some subclasses, the satisfiability problem remains tractable. For instance, it has been shown in [BKB06] that any dependency quantified Horn formula $\Phi \in \text{DQHORN}^*$ of length $|\Phi|$ with free variables, $|\forall|$ universal variables, and an arbitrary number of existential variables can be transformed into a logically equivalent quantified Horn formula of length $O(|\forall| \cdot |\Phi|)$ which contains only existential quantifiers. Moreover, the satisfiability problem for DQHORN* is solvable in time $O(|\forall| \cdot |\Phi|)$.

29.5. Q-Resolution

We now extend the concept of resolution for propositional formulas to quantified Boolean formulas in conjunctive normal form. The idea of resolution is that two clauses are combined into a resolvent, so that the connecting literals disappear. In case of quantified Boolean formulas, these are existential or free literals, and additionally, any universally quantified variable that no longer dominates any existential variable can be deleted. This reduction provides a new possibility for generating the empty clause.

We first introduce the following terminology and notations. Subsequently, we assume that the formulas are in CNF, and therefore in prenex form, and do not contain the Boolean constants 1 and 0. A \forall -literal (universal literal) is a literal over a universally quantified variable. An \exists -literal is a literal over an existentially quantified or a free variable. A \forall -clause is a clause which only contains \forall -literals.

Definition 29.5.1. Ordering of Literals

Let Φ be a quantified Boolean formula, and let L_1 and L_2 be literals in Φ . Then L_1 is less than L_2 ($L_1 < L_2$) if the variable in L_1 occurs in the prefix before the variable of L_2 , or the variable of L_1 is free and the variable of L_2 is bound.

In order to motivate the definition of Q-resolution, we first summarize some observations.

1. If a non-tautological clause of a formula $\Phi \in \text{QCNF}^*$ is a \forall -clause, then Φ is unsatisfiable, and the clause can be replaced by the empty clause, which is false by definition.
2. An existentially quantified formula $\exists y_1 \dots \exists y_n \phi$ in QCNF^* is unsatisfiable if and only if the resolution for propositional formulas leads to the empty clause when only \exists -literals are resolved.
3. Let $(\dots \vee x)$ be a non-tautological clause with a universal variable x and an \exists -literal. If x does not dominate an existential variable, then x can be deleted.
4. Usually, the generation of tautological clauses is of no help in the process of deriving the empty clause, thus we explicitly exclude the generation and application of tautological clauses that are tautological because of \forall -literals.

A restriction of Q-resolution, the Q-unit-resolution, has been introduced in [KKBS87] and was later extended to Q-resolution in [FKKB95].

Definition 29.5.2. Q-Resolution

Let $\Phi = Q\phi$ be a formula in QCNF^* with matrix ϕ and prefix Q . Let ϕ_1 and ϕ_2 be non-tautological clauses of ϕ , where ϕ_1 contains the \exists -literal y , and ϕ_2 contains the literal $\neg y$. We obtain a Q -resolvent σ from ϕ_1 and ϕ_2 by applying the following steps (1) to (3):

1. Eliminate all occurrences of \forall -literals in ϕ_i which are not smaller than an \exists -literal which occurs in ϕ_i for $i = 1, 2$. The resulting clauses are denoted by ϕ'_1 and ϕ'_2 .
2. Eliminate the occurrences of y from ϕ'_1 and the occurrences of $\neg y$ from ϕ'_2 . We obtain ϕ''_1 and ϕ''_2 .

3. $\sigma := \phi_1'' \vee \phi_2''$. If σ is a non-tautological clause, then σ is a resolvent.

A single Q-resolution step is written as $\Phi \mid_{Q\text{-Res}}^1 Q(\phi \wedge \sigma)$ or $\phi \mid_{Q\text{-Res}}^1 \sigma$ or $\Phi \mid_{Q\text{-Res}}^1 \sigma$. We use $\mid_{Q\text{-Res}}^*$ to denote the derivability relation given by the reflexive and transitive closure of $\mid_{Q\text{-Res}}^1$.

Examples

1. $\Phi = \forall x_1 \forall x_2 \exists y((x_1 \vee y) \wedge (x_2 \vee \neg y))$

The formula is unsatisfiable, since the resolvent is a non-tautological \forall -clause: $(x_1 \vee y), (x_2 \vee \neg y) \mid_{Q\text{-Res}}^1 (x_1 \vee x_2)$

2. $\Phi = \forall x_1 \exists y_1 \exists y_2 \forall x_2 \exists y_3((y_1 \vee x_2 \vee y_3) \wedge (y_2 \vee \neg y_3) \wedge (x_1 \vee \neg y_1) \wedge (x_1 \vee \neg y_2))$

A Q-resolution refutation is as follows:

$$(y_2 \vee \neg y_3), (y_1 \vee x_2 \vee y_3) \mid_{Q\text{-Res}}^1 (y_1 \vee y_2 \vee x_2)$$

$$(y_1 \vee y_2 \vee x_2), (x_1 \vee \neg y_1) \mid_{Q\text{-Res}}^1 (x_1 \vee y_2)$$

$$(x_1 \vee y_2), (x_1 \vee \neg y_2) \mid_{Q\text{-Res}}^1 (x_1) \approx \perp.$$

3. $\Phi = \forall x \exists y((x \vee y) \wedge (\neg x \vee \neg y))$

There is no Q-resolvent, because resolution would produce a tautological \forall -clause $(x \vee \neg x)$.

The elimination of universals which do not dominate any existential in the same clause (Item 1. in Definition 29.5.2) has already been presented independently of Q-resolution as a general QBF* simplification rule called universal reduction. We must include it here, because Q-resolution is only refutation complete if universal reduction is applied in each resolution step. In the unsatisfiable formula

$$\Phi = \exists y_1 \forall x_1 \forall x_2 \exists y_2 ((y_1 \vee x_1 \vee y_2) \wedge (\neg y_1 \vee \neg x_1 \vee y_2) \wedge (x_2 \vee \neg y_2))$$

initial universal reduction cannot yield any simplifications, and we cannot resolve on y_1 due to the tautological occurrences of x_1 and $\neg x_1$. If we resolve on y_2 instead, we obtain the two resolvents $(y_1 \vee x_1 \vee x_2)$ and $(\neg y_1 \vee \neg x_1 \vee x_2)$. Now we can only resolve on y_1 if we first apply universal reduction on both resolvents, which produces the simplified clauses (y_1) and $(\neg y_1)$ that subsequently resolve to the empty clause.

The previous example also shows that the exchange lemma for resolution in propositional formulas [KBL99] no longer holds for QBF*. In the example, resolving first on y_2 and then on y_1 leads to the empty clause, whereas direct Q-resolution on y_1 leads to a tautological clause: the process is then *blocked*. Such blockages usually require a detour of several steps.

Lemma 29.5.1. *Let $\Phi = \Pi\phi$ be a formula in QCNF* with matrix ϕ and prefix Π . Suppose that $\Phi \mid_{Q\text{-Res}}^1 \sigma$ for a clause σ , then we have $\Phi \approx \Pi(\phi \wedge \sigma)$.*

Theorem 29.5.2. [FKKB95] *Let $\Phi \in \text{QCNF}^*$, then Φ is unsatisfiable (false) if and only if $\Phi \mid_{Q\text{-Res}}^* \sigma$ for a non-tautological \forall -clause σ .*

Idea of the proof: Proof by induction on k , the number of quantifiers in Φ . Without loss of generality, we consider only closed formulas. Otherwise, we can bind the free variables existentially.

1. $k = 1$ and $\Phi = \exists x\phi$. In that case, the Q-resolution is the same as the ordinary resolution for propositional formulas, and we obtain $\frac{*}{Q-Res} \sqcup$.

2. $k = 1$ and $\Phi = \forall y\phi$. Since Φ is closed and false, the formula contains a non-tautological \forall -clause. That is $\Phi \frac{*}{Q-Res} \sqcup$.

3. $k > 1$ and let $\Phi = \exists y \dots \phi$. Let Φ_0 (respectively Φ_1) be the formula we obtain when we substitute 0 for y (respectively $y = 1$) and reduce the formula accordingly. The formulas Φ_0 and Φ_1 are false. By the induction hypothesis for $k - 1$, it follows that $\Phi_0 \frac{*}{Q-Res} \sigma_0$ and $\Phi_1 \frac{*}{Q-Res} \sigma_1$ for some non-tautological \forall -clauses σ_0 and σ_1 . We obtain $\Phi \frac{*}{Q-Res} \sigma_0$ (respectively $\Phi \frac{*}{Q-Res} \sigma_1$) or simultaneously $\Phi \frac{*}{Q-Res} (y \vee \sigma_0)$ and $\Phi \frac{*}{Q-Res} (\neg y \vee \sigma_1)$. Since y is smaller than the universal variables in σ_0 and σ_1 , we obtain in one Q-resolution step the empty clause.

4. $k > 1$ and $\forall x Q\phi$. Since Φ is false, one of the formulas $\Phi_1 := Q\phi[x/1]$ and $\Phi_0 := Q\phi[x/0]$ must be false. Without loss of generality, suppose Φ_1 is false. By the induction hypothesis, we have $\Phi_1 \frac{*}{Q-Res} \sigma$ for some non-tautological \forall -clause σ . The Q-resolution steps can be carried out in Φ , since we only use clauses which contain neither x nor $\neg x$ or clauses with $\neg x$. That means a non-tautological \forall -clause is a Q-resolvent in Φ . That is, $\Phi \frac{*}{Q-Res} \sigma$ or $\Phi \frac{*}{Q-Res} (\neg x \vee \sigma)$. None of these clauses can be a tautological \forall -clause, since the variable x does not occur in σ .

Suppose $\Phi \frac{*}{Q-Res} \sigma$ for a non-tautological \forall -clause and $\Phi = Q\phi$. Then we have $\Phi \approx Q(\phi \wedge \sigma)$. The latter formula is false.

Most of the resolution restrictions known for propositional formulas can be adapted in a straightforward way to Q-resolution. For instance, unit resolution is refutation complete and correct for propositional Horn formulas. Now we extend the unit resolution to the so called Q-unit resolution. A clause of a quantified CNF formula is called a \exists -unit clause if the clause does not contain more than one \exists -literal (free or existentially quantified literal) besides some universal literals.

Definition 29.5.3. *Q-Unit-Resolution*

Q-unit resolution is Q-resolution, but with the restriction that one of the parent clauses is an \exists -unit clause. If σ is the resolvent from ϕ_1 and ϕ_2 , then we write $\phi_1, \phi_2 \frac{1}{Q-Unit-Res} \sigma$.

Obviously, Q-unit resolution is correct, but not refutation complete for QCNF* formulas. In case of an unsatisfiable formula without \exists -unit clauses, the Q-unit resolution cannot be applied to the formula. Since \exists -unit clauses only refer to the number of free and existential variables, we extend the class of quantified Horn formulas to the following class QEHORN*.

Definition 29.5.4. *QEHORN**

A formula $\Phi \in QCNF^*$ is a *quantified extended Horn formula* if, after eliminating the \forall -literals, the clauses are Horn clauses. The class of all such formulas is denoted by *QEHORN**.

An example of a QEHORN* formula which is not a quantified Horn formula is the following

$$\Phi = \forall x \exists y ((x \vee z \vee \neg y) \wedge (\neg x \vee y)).$$

In the first clause, two literals x and z occur positively. The clause is not a Horn clause. But the clause contains at most one positive \exists -literal (z). The second clause is an \exists -unit clause, because only one \exists -literal is given (y). It can be shown that Q-unit resolution is refutation complete for QEHORN* [FKKB90]. A more restricted version of Q-unit resolution is obtained by restricting Q-unit resolution to positive \exists -unit clauses. Here, the \exists -unit clause contains at most one positive free or existentially quantified literal. That means one of the parent clauses has the form $(w \vee y)$, where y is the free or existential variable and w is the (possibly empty) disjunction of universal literals.

Definition 29.5.5. *Q-Pos-Unit-Resolution (Q-PU-Res)*

Q-pos-unit-resolution is Q-unit-resolution with the additional restriction that in each Q-unit-resolution step, one of the parent clauses must be a positive \exists -unit clause.

Theorem 29.5.3. *Let $\Phi \in \text{QEHORN}^*$, then we have*

$$\Phi \text{ is unsatisfiable if and only if } \Phi \vdash_{\text{Q-Pos-Unit-Res}}^* \perp .$$

For propositional formulas in conjunctive normal form, the length of resolution refutations cannot be bound by a polynomial. That has been shown by a careful analysis of the so-called pigeon hole formulas. For extended quantified Horn formulas, there are also hard formulas for the Q-resolution. Let Φ_k be the following formula ($k \geq 1$):

$$\begin{aligned} \Phi_k = & \exists y_1 \exists y'_1 \forall x_1 \exists y_2 \exists y'_2 \forall x_2 \exists y_3 \exists y'_3 \dots \forall x_{k-1} \exists y_k \exists y'_k \forall x_k \exists y_{k+1} \dots \exists y_{k+k} \\ & ((\neg y_1 \vee \neg y'_1) \wedge \\ & (y_1 \leftarrow x_1, y_2, y'_2) \wedge (y'_1 \leftarrow \neg x_1, y_2, y'_2) \wedge \\ & (y_2 \leftarrow x_2, y_3, y'_3) \wedge (y'_2 \leftarrow \neg x_2, y_3, y'_3) \wedge \\ & \dots \dots \dots \\ & \dots \dots \dots \\ & (y_{k-1} \leftarrow x_{k-1}, y_k, y'_k) \wedge (y'_{k-1} \leftarrow \neg x_{k-1}, y_k, y'_k) \wedge \\ & (y_k \leftarrow x_k, y_{k+1}, \dots, y_{k+k}) \wedge (y'_k \leftarrow \neg x_k, y_{k+1}, \dots, y_{k+k}) \wedge \\ & (x_1 \vee y_{k+1}) \wedge \dots \wedge (x_k \vee y_{k+k}) \wedge (\neg x_1 \vee y_{k+1}) \wedge \dots \wedge (\neg x_k \vee y_{k+k}) \end{aligned}$$

With Q-unit-resolution, there is only a limited number of possible resolvents to choose initially. We can only resolve the clauses in the last row with the clauses in the second to last row. Some possible resolutions are blocked due to tautological occurrences of x_k . For example, we cannot resolve $(y_k \leftarrow x_k, y_{k+1}, \dots, y_{k+k})$ with $(x_k \vee y_{k+k})$, but with $(\neg x_k \vee y_{k+k})$. We obtain positive \exists -unit clauses of the form $(y_k \vee x_1^{\epsilon_1} \vee \dots \vee x_k^{\epsilon_k})$ (respectively $(y'_k \vee x_1^{\epsilon_1} \vee \dots \vee x_k^{\epsilon_k})$) where $x_j^{\epsilon_j} = x_j$ if $\epsilon_j = 1$ and $x_j^{\epsilon_j} = \neg x_j$ if $\epsilon_j = 0$. These can be simplified through universal reduction, because x_k occurs after y_k and y'_k in the prefix. We obtain $(y_k \vee x_1^{\epsilon_1} \vee \dots \vee x_{k-1}^{\epsilon_{k-1}})$ (respectively $(y'_k \vee x_1^{\epsilon_1} \vee \dots \vee x_{k-1}^{\epsilon_{k-1}})$), which can be resolved with the clauses in the third to last row in the formula above.

As the process continues, we can observe that every derivable positive \exists -unit clause with \exists -literal y_i (respectively y'_i) for $1 \leq i \leq k$ has the form $(y_i \vee x_1^{\epsilon_1} \vee \dots \vee x_{i-1}^{\epsilon_{i-1}})$ (respectively $(y'_i \vee x_1^{\epsilon_1} \vee \dots \vee x_{i-1}^{\epsilon_{i-1}})$). These \exists -unit clauses can only be derived in at least 2^{i-1} steps by means of Q-unit-resolution as discussed before. To see why the number of steps is exponential in i , consider the two clauses $\alpha = (y_i \leftarrow x_i, y_{i+1}, y'_{i+1})$ and $\beta = (y'_i \leftarrow \neg x_i, y_{i+1}, y'_{i+1})$. Since x_i occurs with different signs in α and β , different resolvents are blocked in both cases. α must be resolved with a clause $(y_{i+1} \vee x_1^{\epsilon_1} \vee \dots \vee x_{i-1}^{\epsilon_{i-1}} \vee \neg x_i)$, whereas β can only be resolved with a clause $(y_{i+1} \vee x_1^{\epsilon_1} \vee \dots \vee x_{i-1}^{\epsilon_{i-1}} \vee x_i)$, which means we need two different derivations of y_{i+1} as \exists -unit. An analogous observation can be made when resolving on y'_{i+1} . It follows that the derivation of y_1 and y'_1 requires at least 2^k steps. These steps cannot be shortened using Q-resolution.

Theorem 29.5.4. [Flö93] *For any $k \geq 1$, there is a formula $\Phi_k \in \text{QEHORN}$ of length $O(k)$ which is false, and the refutation requires at least 2^k Q-resolution steps.*

For propositional formulas in conjunctive normal form, unit propagation is a powerful technique to reduce the length of the formula. The previous example shows that for QCNF* formulas, propagating unit literals over existentially quantified or free variables may lead to an explosion of possible resolvents and potentially longer clauses. One of the reasons is that the \exists -units may contain universal variables which are not previously contained in the other parent clause.

The satisfiability problem for QEHORN* is PSPACE-complete [Flö93]. The proof of the PSPACE-completeness shows that for closed formulas, we need at most two existential literals per clause. Let QE2-HORN be the set of formulas in which the existential part of the clause is a 2-Horn clause. Then the satisfiability problem remains PSPACE-complete. But if we restrict ourselves to formulas with a fixed number of alternations in the prefix, then the satisfiability problem is coNP-complete. More formally, for $k \geq 2$, the satisfiability problem for QEHORN* formulas with prefix type Π_k (Σ_k respectively) is coNP-complete [FKKB90].

29.6. Quantified Horn Formulas and Q2-CNF

For propositional Horn formulas, the satisfiability problem can be solved in linear time. As we will see, the satisfiability problem for quantified Horn formulas (QHORN*) is also solvable in polynomial time, where the fastest known algorithm requires more than linear, but not more than quadratic time. The models for satisfiable QHORN formulas have a quite simple structure, i.e. they are constants or conjunctions of literals. Quantified Horn formulas with free variables are logically equivalent to propositional Horn formulas, but the length of the shortest Horn formula may be exponential in the length of the quantified formula. For example, let α_n be the following formula:

$$\alpha_n = \bigwedge_{1 \leq j_1, \dots, j_n \leq n} (z \leftarrow z_{1,j_1}, \dots, z_{n,j_n})$$

The formulas α_n have length $O(n^n)$. The length of the shortest CNF formula is not shorter than α_n . The following QHORN* formula is logically equivalent to α_n and has length $O(n^2)$:

$$\exists x_1 \dots \exists x_n ((z \leftarrow x_1, \dots, x_n) \wedge \bigwedge_{1 \leq i, j \leq n} (x_i \leftarrow z_{i,j}))$$

We next present the idea of testing formulas in QHORN* for satisfiability. Since a QHORN* formula with free variables is satisfiable if and only if the existential closure of the formula is true, we only consider closed formulas. Given is the formula

$$\Phi = \exists y_0 \forall x_1 \exists y_1 \dots \forall x_k \exists y_k \phi.$$

We assume that Φ contains no clause consisting only of universal literals and no tautological clause. Since the matrix ϕ is a Horn formula, every clause in Φ belongs to one of the following three classes:

1. P_Φ is the set of clauses with a positive \exists -literal.
2. N_Φ^+ is the set of clauses with a positive \forall -literal.
3. N_Φ is the set of clauses with only negated literals.

One can show that Φ is false if and only if there is a clause φ in $N_\Phi^+ \cup N_\Phi$ such that $\exists y_0 \forall x_1 \exists y_1 \dots \forall x_k \exists y_k (P_\Phi \wedge \varphi)$ is false. That is, clauses with one positive \exists -literal and one clause with only negated \exists -literals are sufficient to decide the satisfiability. We proceed with a case distinction:

Case 1: $\varphi \in N_\Phi$. Then every universal literal occurs only positively in $P_\Phi \wedge \varphi$. Therefore, we can remove all occurrences of universal variables without any effect on the satisfiability. The resulting formula is an existentially quantified formula and satisfiable if and only if the matrix, a propositional Horn formula, is satisfiable. We can apply the linear-time satisfiability algorithm for propositional Horn formulas to that formula.

Case 2: Let $\varphi \in N_\Phi^+$ and let x_i be the positive literal in φ . We can delete all other universal literals, because they occur only negatively in $N_\Phi^+ \cup P_\Phi$. Let $P(i)_\Phi$ be the reduced formula. If the clause φ contains existential literals $\neg y_j$ with $j < i$, then we test whether these variables are derivable units from $P(i)_\Phi$. That can be done by deleting the universal variable x_i in all the clauses of $P(i)_\Phi$ and applying unit propagation to the formula. If not all these \exists -literals y_j are derivable units, the formula Φ is satisfiable.

Otherwise, we remove these \exists -literals $\neg y_j$ from φ .

For the remaining \exists -literals $\neg y_t$ in φ , we have $i \leq t$. For y_t , at most two distinct positive \exists -unit clauses can be derived from $P(i)_\Phi$, namely y_t and $(\neg x_i \vee y_t)$. Because of the completeness of the positive Q-unit resolution for quantified Horn formulas, $\exists y_0 \forall x_1 \exists y_1 \dots \forall x_k \exists y_k (P_\Phi \wedge \varphi)$ is false if and only if all the literals y_t are derivable using positive Q-unit resolution. That can be decided in linear time.

Altogether, that leads to an algorithm for testing the satisfiability. The outline of the algorithm is as follows: At first, we decide whether $P_\Phi \cup N_\Phi$ is satisfiable. If that is not the case, we apply for every universal variable x_i which occurs positively in Φ the procedure explained above. Since the satisfiability problem for propositional Horn formulas is decidable in linear time and unit propagation is a linear-time procedure, we have the following result.

Theorem 29.6.1. [FKKB95] *The satisfiability problem for QHORN* can be decided in time $O(|\forall| \cdot n)$, where n is the length of the formula and $|\forall|$ is the number of universal variables which occur positively in the formula.*

Based on the observation that the satisfiability and equivalence models for formulas in QHORN* have a quite simple structure, the following theorem has been shown [BKBZ05].

Theorem 29.6.2. *Every formula $\Phi \in \text{QHORN}^*$ can be transformed into a logically equivalent and existentially quantified Horn formula in time $O(|\forall| \cdot n)$, where n is the length of the formula and $|\forall|$ is the number of universal variables.*

For propositional Horn formulas, the problem to decide whether two formulas are logically equivalent is solvable in quadratic time. In contrast to the propositional case, the equivalence problem for quantified Horn formulas with free variables is coNP-complete [KBL99].

Theorem 29.6.3. *The consequence problem*

$$\{(\Phi_1, \Phi_2) \mid \Phi_1 \models \Phi_2, \Phi_1, \Phi_2 \in \text{QHORN}^*\}$$

and the equivalence problem

$$\{(\Phi_1, \Phi_2) \mid \Phi_1 \approx \Phi_2, \Phi_1, \Phi_2 \in \text{QHORN}^*\}$$

are coNP-complete.

The proof of the coNP-completeness can be shown by a reduction from the complement of the NP-complete Monotone 3-SAT problem. A formula ϕ is in Monotone 3-SAT if and only if Φ is satisfiable, $\phi = \alpha \wedge \beta \in 3\text{-CNF}$, α contains only positive literals, and β has negative literals. Let be given

$$\begin{aligned} \alpha &= \{(L_{i,1} \vee L_{i,2} \vee L_{i,3}) \mid 1 \leq i \leq n\} \\ \beta &= \{(\neg K_{j,1} \vee \neg K_{j,2} \vee \neg K_{j,3}) \mid 1 \leq j \leq m\}. \end{aligned}$$

where $L_{i,s}$ and $K_{j,t}$ are variables. Now we introduce a new variable y and new variables $x_1, \dots, x_n, z_1, \dots, z_m$. We define

$$\begin{aligned} \Phi_1 &= \exists x_1 \dots \exists x_n \{(y \leftarrow x_1, \dots, x_n)\} \cup \{(x_i \leftarrow L_{i,p}) \mid 1 \leq p \leq 3, 1 \leq i \leq n\} \\ \Phi_2 &= \exists x_1 \dots \exists x_n \exists z_1 \dots \exists z_m \{(y \leftarrow x_1, \dots, x_n, z_j) \mid 1 \leq j \leq m\} \cup \\ &\quad \{(x_i \leftarrow L_{i,p}) \mid 1 \leq p \leq 3, 1 \leq i \leq n\} \cup \{(z_j \leftarrow K_{i,j}) \mid 1 \leq j \leq m\} \end{aligned}$$

Then we have: $\Phi_2 \approx \Phi_1$ if and only if $\Phi_2 \models \Phi_1$ if and only if $\alpha \wedge \beta$ is unsatisfiable. The construction shows that the problems remain coNP-complete for existentially quantified Horn formulas.

For a propositional formula ϕ over the variables x_1, \dots, x_n , a function $f : \text{literals}(\phi) \rightarrow \text{literals}(\phi)$ is called a *renaming* if and only if for all $L \in \text{literals}(\phi)$, we have $f(L) \in \{L, \neg L\}$ and $f(\neg L) \approx \neg f(L)$. The concept of renaming can be extended to quantified Boolean formulas. Let f be a renaming and Φ a QBF* formula, then $f(\Phi)$ is the result of applying the renaming to the matrix while the prefix remains unchanged. For example, let $f(x) = \neg x$, $f(y) = y$ and $\Phi = \forall x \exists y (x \vee y)$ be a formula not in QHORN*. Then we obtain $f(\Phi) = \forall x \exists y f((x \vee y)) = \forall x \exists y (\neg x \vee y)$. The resulting formula is now a quantified Horn formula. Obviously, a renaming preserves the satisfiability of the formulas. Whether a

quantified Boolean formula is renamable into a quantified Horn formula can be decided in linear time by applying the linear-time algorithm for propositional formulas in CNF. Moreover, for any satisfiable Q2-CNF* formula Φ , there is a renaming f with $f(\Phi) \in \text{Q2-HORN}^*$.

By means of renaming, we can decide as follows whether a Q2-CNF* formula is satisfiable. At first decide whether the formula is renamable into a quantified Horn formula. If that is not the case, then the formula is unsatisfiable. Otherwise, we apply the satisfiability algorithm for QHORN* formulas to the renamed formula. But this procedure is not the best one. An alternative algorithm deciding the satisfiability of Q2-CNF* formulas has been established in [APT79]. The idea of the satisfiability algorithm is based on a representation of formulas as graphs. We associate with a formula Φ the graph $G(\Phi)$ as follows:

1. If L is the literal of a unit clause in Φ , then $(\neg L) \rightarrow L$ is an edge in $G(\Phi)$
2. If $(L_1 \vee L_2)$ is a clause in Φ , then $(\neg L_1) \rightarrow L_2$ and $(\neg L_2) \rightarrow L_1$ are edges in $G(\Phi)$
3. Nothing else is an edge in $G(\Phi)$

Lemma 29.6.4. *Let $\Phi \in \text{Q2-CNF}^*$, and let $G(\Phi)$ be the associated graph. Then Φ is false if and only if one of the following conditions holds:*

1. *There is an \exists -variable y and a strongly connected component S of $G(\Phi)$ with $y, \neg y \in S$.*
2. *There is a \forall -variable x and an \exists -variable y with $y < x$, for which there exists a strongly connected component S of $G(\Phi)$ which contains both a literal over x and a literal over y .*
3. *There exist two \forall -literals L_1 and L_2 over distinct variables or one negated and the other unnegated, such that there is a path from L_1 to L_2 in the graph $G(\Phi)$.*

The first condition covers the propositional and the purely existentially quantified case. In condition (2), it is necessary that $y < x$, x does not dominate y in the prefix. The graphs of the formulas $\Phi_1 = \exists y \forall x ((y \vee x) \wedge (\neg y \vee \neg x))$ and $\Phi_2 = \forall x \exists y ((y \vee x) \wedge (\neg y \vee \neg x))$ are identical, but Φ_1 is false and Φ_2 is true. The third condition is necessary for formulas of the form $\Phi = \forall x_1 \forall x_2 \exists y ((\neg x_1 \vee y) \wedge (x_2 \vee \neg y))$. There is a path $\neg x_1 \rightarrow y, y \rightarrow x_2$ from $\neg x_1$ to x_2 .

There is an alternative formulation of condition (3) which is easier to implement.

Condition 3'. There is a strongly connected component S in $G(\Phi)$ which contains two \forall -literals L_1 and L_2 over distinct variables or one negated and the other not, or else there are two strongly connected components S_1 and S_2 in $G(\Phi)$ which are connected, each containing one \forall -literal.

The test for satisfiability can be carried out in linear time, since the computation of the strongly connected components and the test whether or not one of the conditions holds can be performed in linear time.

Theorem 29.6.5. [APT79] *The satisfiability problem for Q2-CNF* is decidable in linear time.*

Every formula in Q2-CNF* is logically equivalent to a propositional 2-CNF formula. The size of an equivalent formula is bound by $4 \cdot r^2$, where r is the

number of free variables, because there are at most this many 2-clauses over r variables.

Lemma 29.6.6. *For every satisfiable formula $\Phi \in \text{Q2-CNF}^*$, we can construct a logically equivalent formula $\Phi' \in \text{Q2-CNF}^*$ in $O((r+1) \cdot n)$ time and linear space such that $|\Phi'| = O(n)$ and Φ' contains no universal variables, where n is the length of Φ and r is the number of free variables.*

As a corollary we obtain the solvability of the equivalence and the consequence problem for Q2-CNF^* .

Lemma 29.6.7. [KBL99]

The consequence problem

$$\{(\Phi_1, \Phi_2) \mid \Phi_1 \models \Phi_2, \Phi_1, \Phi_2 \in \text{Q2-CNF}^*\}$$

and the equivalence problem

$$\{(\Phi_1, \Phi_2) \mid \Phi_1 \approx \Phi_2, \Phi_1, \Phi_2 \in \text{Q2-CNF}^*\}$$

are solvable in quadratic time.

References

- [APT79] B. Aspvall, M. Plass, and R. Tarjan. *A Linear-Time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas*. Information Processing Letters, 8(3):121–123, 1979.
- [Ben05] M. Benedetti. *Evaluating QBFs via Symbolic Skolemization*. Proc. 11th Intl. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'04). Springer LNCS 3452, pp. 285–300, 2005.
- [BKB06] U. Bubeck and H. Kleine Büning. *Dependency Quantified Horn Formulas: Models and Complexity*. Proc. 9th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'06). Springer LNCS 4121, pp. 198–211, 2006.
- [BKBZ05] U. Bubeck, H. Kleine Büning, and X. Zhao. *Quantifier Rewriting and Equivalence Models for Quantified Horn Formulas*. Proc. 8th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'05). Springer LNCS 3569, pp. 386–392, 2005.
- [CKS01] N. Creignou, S. Khanna, and M. Sudan. *Complexity Classifications of Boolean Constraint Satisfaction Problems*. SIAM Monographs on Discrete Applied Mathematics, 2001.
- [CMLBLM06] S. Coste-Marquis, D. Le Berre, F. Letombe, and P. Marquis. *Complexity Results for Quantified Boolean Formulae Based on Complete Propositional Languages*. Journal on Satisfiability, Boolean Modelling and Computation 1:61–88, 2006.
- [Dal97] V. Dalmau. *Some Dichotomy Theorems on Constant-free Boolean Formulas*. Technical Report TR-LSI-97-43-R, Universitat Politècnica de Catalunya, 1997.

- [EST⁺04] U. Egly, M. Seidl, H. Tompits, S. Woltran, and M. Zolda. *Comparing Different Prenexing Strategies for Quantified Boolean Formulas*. Proc. 6th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'03). Springer LNCS 2919, pp. 214–228, 2004.
- [FK06] T. Feder and P. Kolaitis. *Closures and Dichotomies for Quantified Constraints*. Electronic Colloquium on Computational Complexity (ECCC) TR06-160, 2006.
- [FKKB90] A. Flögel, M. Karpinski, and H. Kleine Büning. *Subclasses of Quantified Boolean Formulas*. Proc. 4th Workshop on Computer Science Logic (CSL'90). Springer LNCS 533, pp. 145–155, 1990.
- [FKKB95] A. Flögel, M. Karpinski, and H. Kleine Büning. *Resolution for Quantified Boolean Formulas*. Information and Computation, 117(1):12–18, 1995.
- [Flö93] A. Flögel. *Resolution für quantifizierte Boolesche Formeln*. Dissertation, Universitt Paderborn, 1993.
- [Hem04] E. Hemaspaandra. *Dichotomy Theorems for Alternation-Bound Quantified Boolean Formulas*. ACM Computing Research Repository, Technical Report cs.CC/0406006, 2004.
- [Hen61] L. Henkin. *Some remarks on infinitely long formulas*. In: *Infinistic Methods* (Warsaw, 1961), pp. 167–183, 1961.
- [KBL99] H. Kleine Büning and T. Lettmann. *Propositional Logic: Deduction and Algorithms*. Cambridge University Press, Cambridge, UK, 1999.
- [KBSZ04] H. Kleine Büning, K. Subramani, and X. Zhao. *On Boolean Models for Quantified Boolean Horn Formulas*. Proc. 6th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'03). Springer LNCS 2919, pp. 93–104, 2004.
- [KBSZ07] H. Kleine Büning, K. Subramani, and X. Zhao. *Boolean Functions as Models for QBF*. Journal of Automated Reasoning, 39(1):49–75, 2007.
- [KBZ05] H. Kleine Büning and X. Zhao. *Equivalence Models for Quantified Boolean Formulas*. Proc. 7th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'04), Revised Selected Papers. Springer LNCS 3542, pp. 224–234, 2005.
- [KKBS87] M. Karpinski, H. Kleine Büning, and P. Schmitt. *On the computational complexity of quantified Horn clauses*. Proc. 1st Workshop on Computer Science Logic (CSL'87). Springer LNCS 329, pp. 129–137, 1987.
- [MS72] A. Meyer and L. Stockmeyer. *The Equivalence Problem for Regular Expressions with Squaring Requires Exponential Space*. Proc. 13th Symp. on Switching and Automata Theory, pp. 125–129, 1972.
- [MS73] A. Meyer and L. Stockmeyer. *Word Problems Requiring Exponential Time*. Preliminary Report, Proc. 5th ACM Symp. on Theory of Computing (STOC'73), pp. 1–9, 1973.
- [PRA01] G. Peterson, J. Reif, and S. Azhar. *Lower Bounds for Multiplayer Non-Cooperative Games of Incomplete Information*. Computers

- and Mathematics with Applications, 41(7-8):957–992, 2001.
- [Sav70] W. Savitch. *Relationships Between Nondeterministic and Deterministic Tape Complexities*. Journal of Computer and System Sciences, 4(2):177–192, 1970.
- [SC85] A. Sistla and E. Clarke. *The complexity of propositional linear time logics*. Journal of the ACM, 32(3):733–749, 1985.
- [Sch78] T. Schaefer. *The complexity of satisfiability problems*. Proc. 10th ACM Symp. on Theory of Computing (STOC’78), pp. 1–9, 1978.
- [Sto76] L. Stockmeyer. *The Polynomial-Time Hierarchy*. Theoretical Computer Science, 3(1):1–22, 1976.
- [Tse70] G. Tseitin. *On the Complexity of Derivation in Propositional Calculus*. In A. Silenko (ed.): Studies in Constructive Mathematics and Mathematical Logic, Part II, pp. 115–125, 1970.
- [Wra76] C. Wrathall. *Complete Sets and the Polynomial-Time Hierarchy*. Theoretical Computer Science, 3(1):23–33, 1976.

Chapter 30

Reasoning with Quantified Boolean Formulas

Enrico Giunchiglia, Paolo Marin, and Massimo Narizzano

30.1. Introduction

The implementation of effective reasoning tools for deciding the satisfiability of Quantified Boolean Formulas (QBFs) is an important research issue in Artificial Intelligence and Computer Science. Indeed, QBF solvers have already been proposed for many reasoning tasks in knowledge representation and reasoning, in automated planning and in formal methods for computer aided design. Even more, since QBF reasoning is the prototypical PSPACE problem, the reduction of many other decision problems in PSPACE are readily available (see, e.g., [Pap94]). For these reasons, in the last few years several decision procedures for QBFs have been proposed and implemented, mostly based either on search or on variable elimination, or on a combination of the two.

In this chapter, after a brief recap of the basic terminology and notation about QBFs (Sec. 30.2, but see Part 2, Chapter 29 and 31 for a more extensive presentation), we briefly review various applications of QBF reasoning that have been recently proposed (Section 30.3), and then we focus on the description of the main approaches which are at the basis of currently available solvers for prenex QBFs in conjunctive normal form (CNF) (Section 30.4). Other approaches and extensions to non prenex, non CNF QBFs are briefly reviewed at the end of the chapter.

30.2. Quantified Boolean Logic

Consider a set P of symbols. A *variable* is an element of P . The set of *Quantified Boolean Formulas* (QBFs) is defined to be the smallest set such that

1. if z is a variable, then z is a QBF;
2. if $\varphi_1, \dots, \varphi_n$ are QBFs then also $(\varphi_1 \wedge \dots \wedge \varphi_n)$ and $(\varphi_1 \vee \dots \vee \varphi_n)$ are QBFs ($n \geq 0$);
3. if φ is a QBF then also $\neg\varphi$ is a QBF;

4. if φ is a QBF and z is a variable, then also $\forall z\varphi$ and $\exists z\varphi$ are QBFs.

Other popular propositional connectives, like implication and equivalence, can be defined on the basis of the given ones. In the following, we use TRUE and FALSE as abbreviations for the empty conjunction and the empty disjunction respectively.

In a QBF $Qz\varphi$ with $Q \in \{\forall, \exists\}$, φ is called the *scope* of Qz and Q is a *quantifier binding* z . An *occurrence of a variable z is free in a QBF φ* if it is not in the scope of a quantifier Q binding z . A *variable z is free in a QBF φ* if it has free occurrences. A QBF is *closed* if it has no free occurrences. For example, the QBF

$$\forall y\exists x_2((\neg x_1 \vee \neg y \vee x_2) \wedge (\neg y \vee \neg x_2) \wedge (x_2 \vee ((x_1 \vee \neg y) \wedge (y \vee x_2)))) \quad (30.1)$$

is not closed, since x_1 is free in it. From the above definitions, it is also clear that a QBF without quantifiers is a propositional formula.

A *valuation* is a mapping \mathcal{I} from the set P of variables to $\{\text{TRUE}, \text{FALSE}\}$. \mathcal{I} can be extended to an arbitrary QBF φ as follows:

1. If φ is a variable z , $\mathcal{I}(\varphi) = \mathcal{I}(z)$;
2. If φ is $\neg\psi$, $\mathcal{I}(\varphi) = \text{TRUE}$ iff $\mathcal{I}(\psi) = \text{FALSE}$;
3. If φ is $(\varphi_1 \wedge \dots \wedge \varphi_n)$, $\mathcal{I}(\varphi) = \text{TRUE}$ iff $\forall i : 1 \leq i \leq n, \mathcal{I}(\varphi_i) = \text{TRUE}$;
4. If φ is $(\varphi_1 \vee \dots \vee \varphi_n)$, $\mathcal{I}(\varphi) = \text{TRUE}$ iff $\exists i : 1 \leq i \leq n, \mathcal{I}(\varphi_i) = \text{TRUE}$;
5. If φ is $\exists x\psi$, $\mathcal{I}(\varphi) = \text{TRUE}$ iff $\mathcal{I}(\varphi_x) = \text{TRUE}$ or $\mathcal{I}(\varphi_{\neg x}) = \text{TRUE}$;
6. If φ is $\forall y\psi$, $\mathcal{I}(\varphi) = \text{TRUE}$ iff $\mathcal{I}(\psi_y) = \text{TRUE}$ and $\mathcal{I}(\psi_{\neg y}) = \text{TRUE}$.

If φ is a QBF and z is a variable, φ_z (resp. $\varphi_{\neg z}$) is the QBF obtained by substituting all the free occurrences of z in φ with TRUE (resp. FALSE).

A valuation \mathcal{I} *satisfies* a QBF φ if $\mathcal{I}(\varphi) = \text{TRUE}$. A QBF is *satisfiable* if there exists a valuation satisfying it. Two QBFs φ_1, φ_2 are (*logically*) *equivalent* if every valuation satisfies $((\neg\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \neg\varphi_2))$, and are *equisatisfiable* if they are both satisfiable or both unsatisfiable.

From the above definitions, it is clear that if φ is closed, any two valuations give the same value to φ . Thus, if φ_1 and φ_2 are both closed QBFs

1. φ_1 and φ_2 are satisfiable if and only if they are given the value TRUE by any valuation, and
2. φ_1 and φ_2 are equivalent if and only if they are equisatisfiable.

30.3. Applications of QBFs and QBF reasoning

Deciding the value of a QBF is the prototypical PSPACE complete problem. As such, many decision problems in PSPACE have been shown to be reducible to the task of deciding the satisfiability of a QBF (see, e.g., [Pap94]). Further, the availability of progressively more efficient QBF solvers has recently fostered the definition of effective encodings of various problems as QBFs, and the use of QBF solvers. In the following, we briefly review some of these recent proposals in the areas of automated planning, knowledge representation and reasoning, formal methods.

In the area of automated planning, QBF solvers are used to solve conformant and conditional planning problems in [Rin99a]. The work of [FG00] defines several

encodings of conformant planning problems and presents a procedure mimicking the computation of a search based QBF solver (see also [CGT03]). Turner [Tur02] shows how many other planning and reasoning tasks can be encoded as QBFs, but does not evaluate the effectiveness of the approach. In [GR04, AGS05], various encodings in QBFs of the famous "Connect4" and evader-pursuer problems on a fixed size checkerboard are presented. See also Part 2, Chapter 19 for details (including the presentation of some encodings) about how conditional planning problems can be casted in Quantified Boolean Logic.

In the area of knowledge representation and reasoning, [EETW00] defines the encoding of several reasoning tasks (including autoepistemic, default logic, disjunctive logic programming, and circumscription problems) in QBFs. [PV03] defines a translation of the modal logic K to QBF (see Part 2, Chapter 29 for more on this topic). [EST⁺03] shows how the problem of the evaluation of nested counterfactuals is reducible to a QBF. [DSTW04] shows that the task of modifying a knowledge base K in order to make the result satisfying and consistent with a set of formulas R and C respectively, can be encoded as a QBF φ such that the valuations satisfying φ correspond to belief change extensions in the original approach. In [BSTW05], the authors describe polynomial-time constructible encodings in QBFs of paraconsistent reasoning principles, and advocates the usage of QBF solvers. [Tom03] gives polynomial reductions mapping a given abduction problem into a QBF φ such that the valuations satisfying φ correspond to solutions of the original problem. [OSTW06] uses QBF solvers to check the equivalence of answer-set programs.

In the area of formal methods, the use of QBF for checking the equivalence of partial implementations has been proposed in [SB01, HBS06]. In [AB00] a similar technique is applied to check whether the structural and behavioral descriptions of protocols agree with each other. In [MS04, GNT07] QBF solvers are used to solve vertex eccentricity problems, while [KHD05, DHK05, JB07, MVB07] show how QBF could improve over SAT in various model checking and formal verification tasks. In [BLS03] QBF solvers are applied to verify pipeline processors. In [CKS05] QBF solvers are proposed for the formal verification of interleaving nondeterministic Boolean programs (see also Part 2, Chapter 18). In the area of Field Programmable Gate Array (FPGA) logic synthesis, [LSB05] uses QBFs for determining if a logic function can be implemented in a given programmable circuit.

30.4. QBF solvers

The development of solvers dedicated to the satisfiability problem of QBFs started with the seminal works of [KBKF95, CGS98] and has, since then, attracted more and more attention. Considering the currently available QBF solvers, the vast majority of them focuses on closed QBFs in prenex conjunctive normal form.

A QBF is in *prenex form* if it has the form

$$Q_1 z_1 Q_2 z_2 \dots Q_n z_n \Phi \quad (n \geq 0) \quad (30.2)$$

where

- every Q_i ($1 \leq i \leq n$) is a quantifier,
- z_1, \dots, z_n are distinct variables, and
- Φ is a propositional formula.

In (30.2), $Q_1 z_1 \dots Q_n z_n$ is the *prefix* and Φ is the *matrix*. (30.2) is in *Conjunctive Normal Form (CNF)* if Φ is a conjunction of clauses, where a *clause* is a disjunction of literals. A *literal* is a variable or its negation.

The motivation of this restriction is that any QBF φ can be in linear time transformed into a closed QBF φ' in CNF such that φ and φ' are equisatisfiable. A simple such procedure

1. renames variables in order to make sure that φ does not contain variables bounded by two distinct quantifier occurrences;
2. converts φ in prenex form by moving quantifiers in front of the formula: in the process, care has to be taken in order to ensure that if Qz occurs in the scope of $Q'z'$ in φ , then Qz occurs in the scope of $Q'z'$ also in the resulting formula (Q and Q' are quantifiers, z and z' are variables). In [EST⁺03], the authors define several polynomial prenexing strategies such that the resulting QBF is guaranteed to belong to the lowest possible complexity class in the polynomial hierarchy;
3. converts the matrix in CNF by using a procedure based on renaming, such as those described in [Tse70, PG86, JS04]: the additional variables introduced by the CNF transformation can be existentially quantified in the scope of all the other quantifiers in the formula;
4. assuming $\{x_1, \dots, x_n\}$ ($n \geq 0$) are the free variables in the QBF φ built so far, $\exists x_1 \dots \exists x_n \varphi$ is the final closed, prenex, CNF QBF.

Each of the above three steps can be performed in linear time, and an efficient procedure performs all the above steps with a single traversal of the input QBF (see Chapter 29 for a more detailed and formal discussion).

For example, using the procedure described in [JS04] for converting the matrix in CNF, it is easy to see that (30.1) is equisatisfiable to the QBF

$$\exists x_1 \forall y \exists x_2 \exists x_3 \{ \{ \bar{x}_1, \bar{y}, x_2 \}, \{ \bar{y}, \bar{x}_2 \}, \{ x_2, x_3 \}, \{ x_1, \bar{y}, \bar{x}_3 \}, \{ y, x_2, \bar{x}_3 \} \}, \quad (30.3)$$

in which

- the matrix is represented as a set of clauses (to be interpreted conjunctively),
- each clause is represented as a set of literals (to be interpreted disjunctively),
- \bar{z} stands for $\neg z$.

However, we have to mention that in the process of converting the formula in prenex and CNF form, some relevant information about the structure of both the matrix and the prefix of in input QBF is lost. Using such structural information (of the matrix and of the prefix) can greatly improve performances of the solvers, as shown in [Bie04, Ben05a, GNT07].

For sake of simplicity, in the following, we will restrict our attention to closed QBFs in CNF and prenex form, and keep the above conventions for representing

matrices, clauses and literals. We further assume that each clause in the matrix is non tautological, i.e., that a clause does not contain both a literal l and \bar{l} : Indeed, such clauses can be safely removed from the matrix without affecting the value of the QBF.

For a literal l ,

- $|l|$ is the variable occurring in l ; and
- \bar{l} is the negation of l if l is a variable, and it is $|l|$ otherwise.

We also say that a literal l is *existential* if $\exists|l|$ belongs to the prefix, and it is *universal* otherwise. With these assumptions, if φ is (30.2) and l is a literal with $|l| = z_i$, we redefine φ_l to be the QBF

- whose matrix is obtained from Φ by removing the clauses C with $l \in C$, and by removing \bar{l} from the other clauses; and
- whose prefix is $Q_1 z_1 Q_2 z_2 \dots Q_{i-1} z_{i-1} Q_{i+1} z_{i+1} \dots Q_n z_n$.

Further, we extend the notation to sequences of literals: If $\mu = l_1; l_2; \dots; l_m$ ($m \geq 0$), φ_μ is defined as $(\dots((\varphi_{l_1})_{l_2})\dots)_{l_m}$. For instance, if φ is (30.3), $\varphi_{x_1; x_3}$ is

$$\forall y \exists x_2 \{ \{ \bar{y}, x_2 \}, \{ \bar{y}, \bar{x}_2 \}, \{ y, x_2 \} \}.$$

30.4.1. Solvers based on search

A simple recursive procedure for determining the satisfiability of a QBF φ simplifies φ to φ_z and (or, respectively) $\varphi_{\bar{z}}$ if z is the leftmost variable in the prefix and z is universal (existential, respectively), till either an empty clause or the empty set of clauses are produced: On the basis of the satisfiability of φ_z and $\varphi_{\bar{z}}$, the satisfiability of φ can be determined according to the semantics of QBFs.

There are some simple improvements to this basic procedure.

Let φ be a QBF (30.2). Consider φ .

The first improvement is that we can directly conclude that φ is unsatisfiable if the matrix of φ contains a contradictory clause. A clause C is *contradictory* if it contains no existential literal. An example of a contradictory clause is the empty clause.

The second improvement is based on the fact that in a QBF we can swap two variables in the prefix if they have the same level. In (30.2), the *level of a variable* z_i is $1 +$ the number of expressions $Q_j z_j Q_{j+1} z_{j+1}$ in the prefix with $j \geq i$ and $Q_j \neq Q_{j+1}$. For example, in (30.3), x_2 and x_3 have level 1, \bar{y} has level 2, \bar{x}_1 has level 3. Thus, assuming that z_i and z_1 have the same level in (30.2), (30.2) is logically equivalent to

$$Q_i z_i Q_2 z_2 \dots Q_{i-1} z_{i-1} Q_1 z_1 Q_{i+1} z_{i+1} \dots Q_n z_n \Phi$$

and we can determine the satisfiability of φ on the basis of φ_{z_i} and/or $\varphi_{\bar{z}_i}$. This allows to introduce some heuristics in the choice of the literal for branching.

Finally, if a literal l is unit or monotone in φ , then φ is logically equivalent to φ_l . In (30.2), a literal l is

- *Unit* if l is existential, and, for some $m \geq 0$,

```

0 function Q-DLL( $\varphi, \mu$ )
1   if ((a contradictory clause is in the matrix of  $\varphi_\mu$ )) return FALSE;
2   if ((the matrix of  $\varphi_\mu$  is empty)) return TRUE;
3   if ( $l$  is unit in  $\varphi_\mu$ ) return Q-DLL( $\varphi, \mu; l$ );
4   if ( $l$  is monotone in  $\varphi_\mu$ ) return Q-DLL( $\varphi, \mu; l$ );
5    $l :=$  (a literal at the highest level in  $\varphi_\mu$ );
6   if ( $l$  is existential) return Q-DLL( $\varphi, \mu; l$ ) or Q-DLL( $\varphi, \mu; \bar{l}$ );
7   else return Q-DLL( $\varphi, \mu; l$ ) and Q-DLL( $\varphi, \mu; \bar{l}$ ).

```

Figure 30.1. The algorithm of Q-DLL.

- a clause (l, l_1, \dots, l_m) belongs to Φ , and
- each literal l_i ($1 \leq i \leq m$) is universal and has a lower level than l . The *level of a literal l* is the level of $|l|$.

For example, in a QBF of the form

$$\dots \exists x_1 \forall y_1 \exists x_2 \dots \{\{x_1, y_1\}, \{x_2\}, \dots\},$$

both x_1 and x_2 are unit.

- *Monotone* or *pure* if
 - either l is existential, \bar{l} does not belong to any clause in Φ , and l occurs in Φ ;
 - or l is universal, l does not belong to any clause in Φ , and \bar{l} occurs in Φ .
- For example, in the QBF

$$\forall y_1 \exists x_1 \forall y_2 \exists x_2 \{\{-y_1, y_2, x_2\}, \{x_1, \neg y_2 \neg x_2\}\},$$

the only monotone literals are y_1 and x_1 .

With such improvements, the resulting procedure, called Q-DLL, is essentially the one presented in the work of Cadoli, Giovanardi, and Schaerf [CGS98], which extends DLL in order to deal with QBFs. Figure 30.1 is a simple, recursive presentation of it. In the figure, given a QBF φ ,

1. FALSE is returned if a contradictory clause is in the matrix of φ_μ (line 1); otherwise
2. TRUE is returned if the matrix of φ_μ is empty (line 2); otherwise
3. at line 3, μ is recursively extended to $\mu; l$ if l is unit (and we say that l has been *assigned as unit*); otherwise
4. at line 4, μ is recursively extended to $\mu; l$ if l is monotone (and we say that l has been *assigned as monotone*); otherwise
5. a literal l at the highest level is chosen and
 - If l is existential (line 6), μ is extended to $\mu; l$ first (and we say that l has been *assigned as left split*). If the result is FALSE, $\mu; \bar{l}$ is tried and returned (and in this case we say that \bar{l} has been *assigned as right split*).
 - Otherwise (line 7), l is universal, μ is extended to $\mu; l$ first (and we say that l has been *assigned as left split*). If the result is TRUE, $\mu; \bar{l}$ is tried and returned (and in this case we say that \bar{l} has been *assigned as right split*).

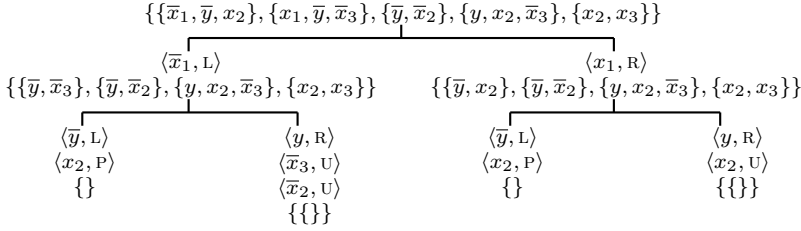


Figure 30.2. The tree generated by $Q\text{-DLL}$ for (30.3). The matrix of (30.3) is shown at the root node, and the prefix is $\exists x_1 \forall y \exists x_2 \exists x_3$. U, P, L, R stand for “unit”, “pure”, “left split”, “right split” respectively, and have the obvious meaning.

Initially, φ is the input formula, and μ is the empty sequence of literals ϵ .

Theorem 1. $Q\text{-DLL}(\varphi, \epsilon)$ returns TRUE if φ is true, and FALSE otherwise.

Given what we have said so far, it is clear that $Q\text{-DLL}$ evaluates φ by generating a semantic tree [Rob68] in which each node corresponds to an invocation of $Q\text{-DLL}$ and thus to an assignment μ . For us,

- an *assignment* (for a QBF φ) is a possibly empty sequence $\mu = l_1; l_2; \dots; l_m$ ($m \geq 0$) of literals such that for each l_i in μ , l_i is unit, or monotone, or at the highest level in $\varphi_{l_1; l_2; \dots; l_{i-1}}$;
- the (*semantic*) *tree* representing a run of $Q\text{-DLL}$ on φ is the tree
 - having a node μ for each call to $Q\text{-DLL}(\varphi, \mu)$; and
 - an edge connecting any two nodes μ and $\mu; l$, where l is a literal.

Any tree representing a run of $Q\text{-DLL}$ has at least the node corresponding to the empty assignment ϵ .

As an example of a run of $Q\text{-DLL}$, consider the QBF (30.3). For simplicity, assume that the literal returned at line 5 in Figure 30.1 is the negation of the first variable in the prefix which occurs in the matrix of the QBF under consideration. Then, the tree searched by $Q\text{-DLL}$ when φ is (30.3) can be represented as in Figure 30.2. In the figure:

- Each node is labeled with the literal assigned by $Q\text{-DLL}$ in order to extend the assignment built so far. Thus, the assignment corresponding to a node is the sequence of labels in the path from the root to the node. For instance, the assignment corresponding to the node with label \bar{x}_3 is $\bar{x}_1; y; \bar{x}_3$.
- When literals are assigned as unit or monotone, the corresponding nodes are aligned one below the other. Further for each assigned literal l , we also show whether l has been assigned as unit, monotone, left or right split by marking it as U, P, L, R respectively.
- When l has been assigned as a left or right split, we also show the matrix of $\varphi_{\mu; l}$, where μ is the sequence of literals assigned before l .
- When the node μ is a leaf, then the matrix of φ_{μ} is either empty (in which case we write “ $\{\}$ ” below the node), or it contains a contradictory clause (in which case we write “ $\{\{\}\}$ ” below the node).

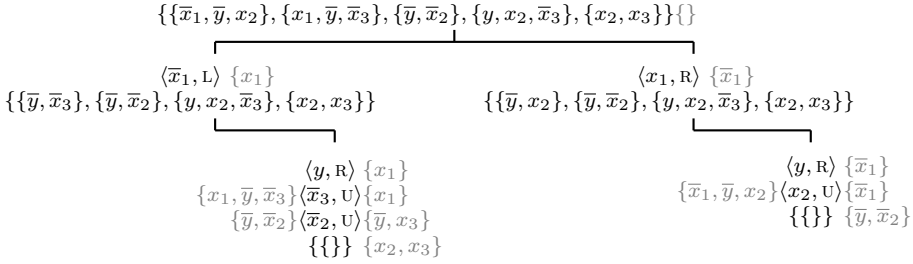


Figure 30.3. The clause resolution corresponding to the tree generated by *Q-DLL* for (30.3). The prefix is $\exists x_1 \forall y \exists x_2 \exists x_3$.

Considering Figure 30.2, it is easy to see that *Q-DLL* would correctly return FALSE, meaning that (30.3) is false.

As in SAT, there is a close correspondence between the search tree explored by *Q-DLL* and a resolution proof showing the (un)satisfiability of the input formula, and this correspondence lays down the foundations for incorporating nogood and good learning in *Q-DLL* [GNT06].

Consider a QBF φ . Assume φ is unsatisfiable and let Π be the search tree explored by *Q-DLL*(φ, ϵ). Then, we can restrict our attention to the *minimal false subtree* of Π , i.e., to the tree obtained from Π by deleting the subtrees starting with a left split on a universal literal: These subtrees are originated from “wrong choices” when deciding which branch to explore first. In the minimal false subtree Π' of Π , all the leaves terminate with the empty clause, and we can associate with each node of Π' a clause pretty much in the same way as in SAT: The leaves of the proof tree associated to the search tree are clauses that generated the empty clause, and the clauses associated to the internal nodes are obtained by resolving the clauses associated to children nodes in the search tree (a literal l assigned as unit can be considered as assigned as right split: performing a left split on \bar{l} would immediately generate the empty clause and this node would be a leaf of the search tree).

For instance, if φ is (30.3), Figure 30.3 shows the minimal false subtree of *Q-DLL*’s computation, and the clause associated to each node. In the figure,

- the clause associated with each node is written in gray and to the right of the node itself;
- when a node corresponds to the assignment of a unit literal l , a clause of φ which causes l to be unit at that node (used in the corresponding clause resolution) is written in gray and to the left of the node.

Indeed, the clause associated to the root node ϵ of the figure, is the empty one, meaning that (30.3) is unsatisfiable.

There are however two differences between the QBF and the SAT cases. The first one, is that Q (clause) resolution [KBKF95] is used instead of plain resolution.

Q (clause) resolution (on a literal l) is the rule

$$\frac{C_1 \quad C_2}{\min(C)} \quad (30.4)$$

where

- l is an existential literal;
- C_1, C_2 are two clauses such that $\{l, \bar{l}\} \subseteq (C_1 \cup C_2)$, and for no literal $l' \neq l$, $\{l', \bar{l}'\} \subseteq (C_1 \cup C_2)$;
- C is $(C_1 \cup C_2) \setminus \{l, \bar{l}\}$;
- $\min(C)$ is the clause obtained from C by removing the universal literals whose level is lower than the level of all the existential literals in C .

C_1 and C_2 are the *antecedents*, and $\min(C)$ is the *resolvent* of the rule.

The second difference is that the process of associating a clause to each node may require more than a single Q -resolution. Indeed, some clause resolutions can be blocked because of universal variables occurring both as y and \bar{y} in the clauses to be used for the resolution. Consider for instance the QBF:

$$\exists x_1 \exists x_2 \forall y \exists x_3 \{ \{x_1, \bar{x}_3\}, \{\bar{x}_2, \bar{y}, x_3\}, \{x_2, y, x_3\}, \{\bar{x}_1, x_3\} \}. \quad (30.5)$$

Then, Q -DLL may explore the following branch:

$$\begin{array}{l} \langle \bar{x}_1, L \rangle \\ \{x_1, \bar{x}_3\} \langle \bar{x}_3, U \rangle \\ \{\bar{x}_2, \bar{y}, x_3\} \langle \bar{x}_2, U \rangle \dots \\ \{\{\}\} \{x_2, y, x_3\} \end{array} \quad (30.6)$$

where it is not possible to perform the clause resolution associated with the node having label $\langle \bar{x}_2, U \rangle$. As in the example, a clause resolution (30.4) may be blocked only because of some “blocking” universal literal l

- with both l and \bar{l} not in μ , and
- with $l \in C_1$ and $\bar{l} \in C_2$.

Since both C_1 and C_2 are in minimal form, this is only possible if both C_1 and C_2 contain an existential literal l'

- having level less than or equal to the level of all the other literals in the clause; and
- assigned as unit.

Then, the obvious solution is to get rid of the blocking literals l in C_1 (or in C_2) by resolving away from C_1 (or from C_2) the existential literals with a level lower than the level of l .

In our example, if φ is (30.5) and with reference to the deduction in (30.6), the blocked clause resolution is the one associated with the node $\bar{x}_1; \bar{x}_3; \bar{x}_2$. Indeed, the two clauses $C_1 = \{x_2, y, x_3\}$ and $C_2 = \{\bar{x}_2, \bar{y}, x_3\}$ cannot be resolved on x_2 because of the occurrences of y and \bar{y} in the two clauses. Considering the first clause $\{x_2, y, x_3\}$, since the level of y is less than the level of x_2 and the clause is in minimal form, there must be another existential literal (in our case x_3) having

level less than the level of y and assigned as unit: By resolving $\{x_2, y, x_3\}$ with the clause $\{x_1, \bar{x}_3\}$ (which causes the assignment of \bar{x}_3 as unit) we get the resolvent $C_3 = \min(\{x_1, x_2, y\}) = \{x_1, x_2\}$ which can be now resolved with C_2 on x_2 . Thus, the clause associated with each node is:

$$\begin{array}{l} \langle \bar{x}_1, L \rangle \{x_1\} \\ \{x_1, \bar{x}_3\} \langle \bar{x}_3, U \rangle \{x_1\} \\ \{\bar{x}_2, \bar{y}, x_3\} \langle \bar{x}_2, U \rangle \{x_1, \bar{y}, x_3\} \\ \{\{\}\} \{x_1, x_2\} \text{ (From } \{x_2, y, x_3\} \{x_1, \bar{x}_3\}\text{)}. \end{array}$$

Notice that the choice of eliminating the blocking literal y in C_1 while keeping C_2 the same, is arbitrary. Indeed, we could eliminate the blocking literal \bar{y} in C_2 and keep C_1 . In the case of the deduction in (30.6), this amounts to eliminate the universal literal \bar{y} in $\{\bar{x}_2, \bar{y}, x_3\}$: By resolving this clause with $\{x_1, \bar{x}_3\}$ on x_3 , we get the resolvent $\{x_1, \bar{x}_2\}$, which leads to the following legal deduction:

$$\begin{array}{l} \langle \bar{x}_1, L \rangle \{x_1\} \\ \{x_1, \bar{x}_3\} \langle \bar{x}_3, U \rangle \{x_1\} \\ \text{(From } \{\bar{x}_2, \bar{y}, x_3\}, \{x_1, \bar{x}_3\}\text{)} \{x_1, \bar{x}_2\} \langle \bar{x}_2, U \rangle \{x_1, y, x_3\} \\ \{\{\}\} \{x_2, y, x_3\}. \end{array}$$

Variants of the above procedures are described in [GNT02, Let02, ZM02a]. In [GNT06], it is proved that we can always associate a clause C (resulting from a sequence of clause resolutions) to the node φ_μ of the minimal false subtree explored by $Q\text{-DLL}$: The clause C is such that for each existential literal $l \in C$, \bar{l} is in μ . Such clauses can be learned as nogoods, pretty much as in SAT.

If φ is satisfiable the situation is the dual one, except that we have to consider terms or cubes instead of clauses. A *term* or *cube* is a conjunction of literals. Terms are associated to the *minimal true subtree* Π' explored by $Q\text{-DLL}$, i.e., to the tree obtained from Π by deleting the subtrees starting with a left split on an existential literal. In more details, to each node φ_μ of Π' we associate a term T such that for each universal literal $l \in T$, \bar{l} is in μ . Such terms can be learned as goods, to be treated as in disjunction with the matrix of φ . Because of the learned goods, also universal literals can be assigned as unit at any point of the search tree. Assuming the current assignment is μ , a universal literal l can be assigned as *unit* if there exists a learned term T such that for each literal $l' \in T$, \bar{l}' is not in μ ; whose only unassigned universal literal is l while all the other unassigned literals have level lower than l . The process of associating terms to each node of Π' starts by associating the conjunction of the literals in μ when the matrix of φ_μ becomes empty. Then, the situation is analogous to the previous case, except that Q term resolution has to be used. *Term resolution (on a literal l)* is the rule

$$\frac{T_1 \quad T_2}{\min(T)}$$

where

- l is an universal literal;
- T_1, T_2 are two terms such that $\{l, \bar{l}\} \subseteq (T_1 \cup T_2)$, and for no literal $l' \neq l$, $\{l', \bar{l}'\} \subseteq (T_1 \cup T_2)$;

- T is $(T_1 \cup T_2) \setminus \{l, \bar{l}\}$;
- $\min(T)$ is the term obtained from T by removing the existential literals whose level is lower than the level of all the universal literals in T .

As before, some term resolution may be blocked, but this situation can be solved by getting rid of the existential blocking literals by performing term resolutions on one of the antecedents. See [GNT02, Let02, ZM02b, GNT06] for more details, and [GNT04] for a discussion about the interactions and problems related to the implementation of learning in the presence of monotone literal fixing.

Given the above, it is possible to associate a clause/term resolution deduction to each branch of the search tree explored by Q - DLL . The resolvents of such deductions can be learned as in SAT. Further, the Unique Implication Point (UIP) mechanism that has shown to be very effective in SAT (see [MSS96] for a presentation of UIP based learning and [ZMMM01] for a comparison of different UIP based learning mechanisms), can be generalized to the QBF case in a simple way. Assume that we are backtracking on a literal l assigned at decision level n , where the *decision level* of a literal is the number of branching nodes before l . The clause/term corresponding to the reason for the current conflict (resp. solution) is learned if and only if:

1. l is existential (resp. universal),
2. all the assigned literals in the reason except l , are at a decision level strictly smaller than n , and
3. there are no unassigned universal (resp. existential) literals in the reason that are before l in the prefix.

Under the above conditions, it is possible to backjump to the node at the maximum decision level among the literals in the reason, excluding l , and to assign l (resp. \bar{l}) as unit.

Also the ideas underlying the heuristic to be used for selecting the next literal to branch on, can be generalized from the SAT case. However, it is also clear that the effectiveness of any devised heuristic not only depends on the structure of the matrix of the QBFs, but also and in a crucial way, on the structure of the prefix. Indeed, QBFs range from formulas like

$$\exists x_1 \forall x_2 \exists x_3 \dots \forall x_{n-1} \exists x_n \Phi \quad (30.7)$$

to formulas like

$$\exists x_1 \exists x_2 \dots \exists x_m \Phi, \quad (30.8)$$

i.e., to SAT instance. If we consider QBFs of the type (30.7) then it is likely that the heuristic is almost useless: unless an atom $|l|$ is removed from the prefix because l is either unit or monotone, the atom to pick at each node is fixed. On the other hand, considering QBFs of the sort (30.8), we know from the SAT literature that nontrivial heuristics are essential to reduce the search space. In practice, QBF instances lay between the extremes marked by (30.7) and (30.8), and instances like (30.7) are fairly uncommon, particularly on QBFs encoding real-world problems. Because of this, the scoring mechanisms devised for SAT in order to decide which literal is best to branch on, are applied also to the QBF case. However, it is clear that the literal with the highest score can be selected

only if it has also the highest level among the unassigned literals. These ideas can be effectively implemented by arranging literals in a priority queue according to (i) the prefix level of the corresponding atom, and (ii) their score. In this way, atoms at prefix level i are always assigned before atoms at prefix level $j > i$ no matter the score, and atoms with the same prefix level are assigned according to their score.

The limitation to branch on literals at the highest level is one of the drawbacks of search based procedures. [Rin99b] proposed techniques based on the inversion of quantifiers, e.g., on assigning universal literals not at the highest level. Indeed, assigning a universal (resp. existential) literal not at the highest level corresponds to weakening (resp. strengthening) the satisfiability of the QBF: If the resulting QBF is unsatisfiable (resp. satisfiable), so it is the original one. These ideas have been proposed and used also in [CSGG02, SB05]. In [CSGG02] a SAT solver is called at each recursive call on the SAT formula obtained by removing all the universal literals from the matrix (this corresponds to consider the QBF in which all the universal quantifiers are pushed down the prefix till level 1): If the SAT formula is satisfiable, this is also the case for the original QBF. Along the same lines, [SB05] calls a SAT solver on the matrix of the QBF (this corresponds to consider the QBF in which all the universal quantifiers are changed to existential ones): If the SAT formula is unsatisfiable, so is also the QBF; in the case an assignment satisfying the matrix is found, such assignment is used to guide the search in the QBF solver.

Beside the above, [GN⁺04] presents lazy data structures for unit and monotone literal detection. [GNT01, GNT03] show how *Q-DLL* can be extended with conflict and solution backjumping. [GHR03] combines Conflict and Solution Directed Backjumping (CSBJ) with a Stochastic Local Search procedure: The result is a procedure which in some cases is unable to determine the (un)satisfiability of the QBF (in which cases it returns “Unknown”), but otherwise the result is guaranteed to be correct. [CMBL05] presents a branching heuristics promoting renamable Horn formulas. [SB06] proposes the use of binary clause reasoning and hyper-resolution.

30.4.2. Solvers based on variable elimination

An alternative approach consists in eliminating variables till the formula contains the empty clause or becomes empty. Elimination of variables can be performed in QBF on the basis that for any QBF (non necessarily in prenex CNF form) φ , $\exists x\varphi$ and $\forall y\varphi$ are logically equivalent to $(\varphi_x \vee \varphi_{\bar{x}})$ and $(\varphi_y \wedge \varphi_{\bar{y}})$ respectively.

The main problem of this approach is that at each step the formula can double its size. There are however several ways to address these issues. The first one, is that if a variable is unit or monotone then it can be simplified as we have seen in the previous section.

Further, in the case of universal variables, each clause C can be replaced with $\min(C)$, and we can avoid the duplication of the clauses not in their minimal scope. Given a universal variable y , a clause C is in the *minimal scope of y* if y occurs in C , or if C is y -connected to a clause already determined to be in the minimal scope of y . Two clauses are y -connected if there exists an existential

variable with level lower than y occurring in both clauses. For example, in (30.3) the minimal scope of y consists of all the clauses in the matrix. In other cases however, the minimal scope can be (significantly) smaller, as in

$$\forall y \exists x_1 \exists x_2 \exists x_3 \{ \{y, x_1\}, \{\bar{y}, \bar{x}_1\}, \{x_2, x_3\}, \{\bar{x}_2, \bar{x}_3\} \}$$

in which the minimal scope of y are just the first two clauses in the matrix. The expansion of a variable y is obtained by

1. adding a variable z' for each variable z with level lower than the level of y and occurring in minimal scope of y ,
2. quantifying each variable z' in the same way and at the same level of the variable z ,
3. for each C in the minimal scope of y , adding a new clause C' obtained from C by substituting the newly introduced variables z' to z ,
4. considering the newly introduced clauses, those containing \bar{y} are eliminated and y is eliminated from the others.
5. considering the clauses in the original minimal scope of y , those containing y are eliminated and \bar{y} is eliminated from the others.

For example, the expansion of the universal variable y in (30.3) yields the SAT formula:

$$\exists x_1 \exists x_2 \exists x_3 \exists x'_2 \exists x'_3 \{ \{x_2, x_3\}, \{x_2, \bar{x}_3\}, \{\bar{x}_1, x'_2\}, \{\bar{x}'_2\}, \{x'_2, x'_3\}, \{x_1, \bar{x}'_3\} \}.$$

which can be determined to be unsatisfiable by a call to a SAT solver. In order to determine the cost of such expansion, for a literal l let S_l be the set of clauses C with $l \in C$, $o(l)$ be $|S_l|$ and $s(l)$ be the sum of the sizes of the clauses in S_l (i.e., $\sum_{C \in S_l} |C|$). Then, if $\text{minscope}(y)$ is the size of the clauses in the minimal scope of y , after y expansion, the size of the matrix of the QBF increases or decreases by

$$\text{minscope}(y) - (s(y) + s(\bar{y}) + o(y) + o(\bar{y})). \quad (30.9)$$

Notice that when the variable y occurs in all the clauses in its minimal scope, then $\text{minscope}(y) = s(y) + s(\bar{y})$ and (30.9) is negative, i.e., the matrix of the QBF shrinks after the expansion.

Alternatively to the expansion of a universal variable y , we can eliminate the existential variables in its minimal scope. In general, the elimination of an existential variable x is performed only when x occurs in clauses in which all the other literals have level greater than or equal to the level of x . If this is the case, we say that x is an *innermost* variable (notice that all the variables with level 1 are innermost). If x is an innermost variable, we can eliminate x by replacing S_x and $S_{\bar{x}}$ with the clauses obtained by resolving each clause in S_x with each clause in $S_{\bar{x}}$ on x , i.e., with the clauses in

$$\{C \cup C' : C \cup \{x\} \in S_x, x \notin C, C' \cup \{\bar{x}\} \in S_{\bar{x}}, \bar{x} \notin C'\}.$$

For example, the elimination of the innermost variable x_3 from (30.3) yields the QBF:

$$\exists x_1 \forall y \exists x_2 \{ \{\bar{x}_1, \bar{y}, x_2\}, \{\bar{y}, \bar{x}_2\}, \{x_1, \bar{y}, x_2\}, \{y, x_2\} \}. \quad (30.10)$$

```

0 function Q-DP( $\varphi$ )
1   if ((a contradictory clause is in the matrix of  $\varphi$ ) return FALSE;
2   if ((the matrix of  $\varphi$  is empty)) return TRUE;
3   if ( $l$  is unit in  $\varphi$ ) return Q-DP( $\varphi_l$ );
4   if ( $l$  is monotone in  $\varphi$ ) return Q-DP( $\varphi_l$ );
5    $z :=$  (a variable in  $\varphi$  having level  $\leq 2$ );
6   if ( $z$  is existential) return Q-DP(resolve( $z, \varphi$ ));
7   else return Q-DP(expand( $z, \varphi$ )).

```

Figure 30.4. The algorithm of Q-DP.

After the elimination of a variable x , the size of the resulting formula can increase or decrease by

$$o(x) \times (s(\bar{x}) - o(\bar{x})) + o(\bar{x}) \times (s(x) - o(x)) - (s(x) + s(\bar{x})), \quad (30.11)$$

where

1. the first (resp. second) addendum takes into account the fact that each clause containing x (resp. \bar{x}) has to be resolved with all the clauses in $S_{\bar{x}}$ (resp. S_x); and
2. the last addendum takes into account the fact that all the clauses in S_x and $S_{\bar{x}}$ can be removed.

From (30.11) it is easy to see that there are cases in which the formula can shrink, i.e., in which (30.11) is negative. This is always the case, e.g., when $o(x) = 1$ and $s(x) = 2$, as it is the case for x_3 in (30.3). Further, in practice the size of the resulting formula may result to be smaller than the one predicted by (30.11). For instance, the size of matrix of (30.10) is 10, while the one predicted by (30.11) is 11. It is however easy to have cases in which (30.11) is positive and the size of the resulting formula is exactly the one predicted by (30.11), e.g., when $o(x) > 1$, $s(x) > 2$ and the variables in S_x are distinct from the variables in $S_{\bar{x}}$.

Figure 30.4 presents a high level description of a recursive procedure incorporating the above ideas. In the figure, given a QBF φ , lines (1)-(4) are analogous to the ones in Figure 30.1, while at line (5)

1. either an existential variable z at level 1 is chosen and then the call $resolve(z, \varphi)$ eliminates z by resolution (line (6)).
2. or a universal variable z at level 2 is chosen and then the call $expand(z, \varphi)$ eliminates z by expansion (line (7)).

If φ is a SAT formula, Q-DP behaves as the Davis Putnam procedure [DP60].

Theorem 2. Q-DP(φ) returns TRUE if φ is true, and FALSE otherwise.

For example, if φ is (30.3), Q-DP(φ) will return FALSE by, e.g., eliminating x_3 and obtaining (30.10), and then eliminating x_2 and obtaining

$$\exists x_1 \forall y \{ \{ \bar{x}_1, \bar{y} \}, \{ x_1, \bar{y} \} \},$$

equivalent to

$$\exists x_1 \{ \{ \bar{x}_1 \}, \{ x_1 \} \},$$

which is obviously unsatisfiable.

Alternatively, the expansion of y in (30.3) yields the SAT formula:

$$\exists x_1 \exists x_2 \exists x_3 \exists x'_2 \exists x'_3 \{ \{x_2, x_3\}, \{x_2, \bar{x}_3\}, \{\bar{x}_1, x'_2\}, \{\bar{x}'_2\}, \{x'_2, x'_3\}, \{x_1, \bar{x}'_3\} \}$$

which can be determined to be unsatisfiable by a call to a SAT solver, or by repeated applications of variable eliminations till the empty clause is produced.

As for Q - DLL , there are many variable elimination procedures which are variations to the above presented procedure. For example, in Figure 30.4, only universal variables y at level ≤ 2 are chosen for expansion: The restriction to level ≤ 2 is not necessary and in [BKB07] it is shown that variables z which occur only negatively or only positively within clauses in the minimal scope of y do not need to be duplicated and therefore do not propagate y -connectivity. In [Ben04, Bie04] a SAT solver is called as soon as the formula does not contain universal variables. In [Ben04, Bie04], existential variables get first eliminated if they are detected to appear in a binary equivalence: Indeed, if two clauses $\{\bar{l}, x\}$ and $\{l, \bar{x}\}$ are in the matrix of QBF, assuming x has a level not greater than the level of l , x can be safely substituted by l . In [Ben04, Bie04, PV04], simplification mechanisms are defined and used in order to remove subsumed clauses: a clause C is *subsumed* if there exists another clause C' with $C' \subset C$. Considering the way variables are selected for elimination, in [Ben04, Bie04] the choice takes into account the size of the resulting formula as estimated by equations (30.9) and (30.11); in [PV04], universal variables are never expanded since the variables being eliminated are always at level 1. Further, these procedures differ also for the way they represent the matrix. In [Ben04] the QBF is first Skolemized: Each clause corresponds to a disjunction of Boolean functions and is represented via Binary Decision Diagrams (BDDs) [Bry92]. In [PV04] clauses are represented and manipulated as BDDs and ZDDs. In [Bie04] a clause is represented as a set of literals.

The procedure described in [PBZ03] is somehow different from Q - DP though it is based on the elimination of variables from the matrix of the QBF. Given a QBF φ , the basic idea of the procedure is to replace a quantified subformula φ' of φ with a logically equivalent formula φ'' without quantifiers: φ''

1. is in CNF since it corresponds to the conjunction of the negation of the valuations falsifying φ' ; and
2. can be computed using a SAT solver as back-engine if all the quantifiers in φ' are of the same type, or using a QBF solver as back-engine in the general case.

Thus, in [PBZ03], more than a variable can be eliminated in a single step, and these variables can be of different type. Indeed, the selection of the subformula φ' which corresponds to the set V of variables to be eliminated is crucial: If S is the union of the minimal scopes of the variables in V , the intuition for the selection of V is to

1. try to minimize the set of variables in S and not in V , and, at the same time,
2. try to maintain V reasonably small in order to be able to enumerate the valuations falsifying φ' .

30.5. Other approaches, extensions and conclusions

Given a QBF, it is indeed possible to combine the techniques described in the previous sections for solving it. This is what it has been proposed in [Ben05b], where variable elimination and search techniques can be alternated inside one solver. In [PT07] a portfolio of solvers is considered, and the best one is selected using machine learning techniques. [DLMS02] shows how it is possible to encode QBFs in the language of the model checker NuSMV [CCG⁺02], and then use it as engine.

Extensions to the above procedures in order to deal with non prenex, non CNF QBFs have been proposed. When dealing with solvers based on variable elimination, we already mentioned that it is useful to compute the minimal scope of variables and this indeed corresponds to deal with non prenex QBFs, see [Bie04, Ben05a, PV04, PBZ03]. In [GNT07] it is shown that basic *Q-DLL* search procedures can be extended to take into account the quantifier structure, and that substantial speed ups can be obtained. Finally, non prenex QBFs are naturally handled by solvers based on Skolemization of the input QBF, which naturally allow for handling the more general Henkin's branching quantifiers [Hen61].

Non clausal QBF solvers have been proposed in [Zha06, SAG⁺06, ESW06]. In more details, the first two papers show that non clausal formulas can be more naturally encoded in both conjunctive and disjunctive normal forms allowing at the same time for substantial speed ups. The last paper presents a procedure which can be directly applied to formulas in negation normal form.

Acknowledgments

This work is partially supported by the Italian Ministero dell'Università e della Ricerca.

References

- [AB00] A. Abdelwaheb and D. Basin. Bounded model construction for monadic second-order logics. In *12th International Conference on Computer-Aided Verification (CAV'00)*, number 1855 in Lecture Notes in Computer Science, pages 99–113, Chicago, USA, July 2000. Springer-Verlag.
- [AGS05] C. Ansótegui, C. P. Gomes, and B. Selman. The achilles' heel of qbf. In *Proc. AAAI*, pages 275–281, 2005.
- [Ben04] M. Benedetti. Evaluating qbfs via symbolic skolemization. In *Proc. LPAR*, pages 285–300, 2004.
- [Ben05a] M. Benedetti. Quantifier Trees for QBFs. In *8th International Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*, volume 3569 of *Lecture Notes in Computer Science*. Springer Verlag, 2005.
- [Ben05b] M. Benedetti. skizzo: A suite to evaluate and certify qbfs. In *Proc. CADE*, pages 369–376, 2005.
- [Bie04] A. Biere. Resolve and expand. In *Proc. SAT*, pages 59–70, 2004.

- [BKB07] U. Bubeck and H. Kleine Büning. Bounded universal expansion for preprocessing qbf. In *Proc. SAT*, pages 244–257, 2007.
- [BLS03] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Convergence testing in term-level bounded model checking. In *Proc. CHARME*, pages 348–362, 2003.
- [Bry92] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
- [BSTW05] P. Besnard, T. Schaub, H. Tompits, and S. Woltran. Representing paraconsistent reasoning via quantified propositional logic. In L. E. Bertossi, A. Hunter, and T. Schaub, editors, *Inconsistency Tolerance*, pages 84–118, 2005.
- [CCG⁺02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An open-source tool for symbolic model checking. In *Proc. CAV*, 2002.
- [CGS98] M. Cadoli, A. Giovanardi, and M. Schaerf. An algorithm to evaluate quantified Boolean formulae. In *Proc. AAAI*, 1998.
- [CGT03] C. Castellini, E. Giunchiglia, and A. Tacchella. SAT-based planning in complex domains: Concurrency, constraints and nondeterminism. *Artificial Intelligence*, 147(1-2):85–117, July 2003.
- [CKS05] B. Cook, D. Kroening, and N. Sharygina. Symbolic model checking for asynchronous Boolean programs. In *Proc. SPIN*, pages 75–90, 2005.
- [CMBL05] S. Coste-Marquis, D. L. Berre, and F. Letombe. A branching heuristics for quantified renamable horn formulas. In *SAT*, pages 393–399, 2005.
- [CSGG02] M. Cadoli, M. Schaerf, A. Giovanardi, and M. Giovanardi. An algorithm to evaluate quantified Boolean formulae and its experimental evaluation. *Journal of Automated Reasoning*, 28:101–142, 2002.
- [DHK05] N. Dershowitz, Z. Hanna, and J. Katz. Bounded model checking with qbf. In *SAT*, pages 408–414, 2005.
- [DLMS02] F. M. Donini, P. Liberatore, F. Massacci, and M. Schaerf. Solving QBF by SMV. In *Proc. KR*, pages 578–592, 2002.
- [DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.
- [DSTW04] J. P. Delgrande, T. Schaub, H. Tompits, and S. Woltran. On computing belief change operations using quantified boolean formulas. *Journal of Logic and Computation*, 14(6):801–826, 2004.
- [EETW00] U. Egly, T. Eiter, H. Tompits, and S. Woltran. Solving advanced reasoning tasks using Quantified Boolean Formulas. In *Proceedings of the 7th Conference on Artificial Intelligence (AAAI-00) and of the 12th Conference on Innovative Applications of Artificial Intelligence (IAAI-00)*, pages 417–422, Menlo Park, CA, July 30– 3 2000. AAAI Press.
- [EST⁺03] U. Egly, M. Seidl, H. Tompits, S. Woltran, and M. Zolda. Comparing different prenexing strategies for quantified Boolean formulas. In *SAT*, pages 214–228, 2003.

- [ESW06] U. Egly, M. Seidl, and S. Woltran. A solver for qbfs in nonprenex form. In *ECAI*, pages 477–481, 2006.
- [FG00] P. Ferraris and E. Giunchiglia. Planning as satisfiability in simple nondeterministic domains. In *AIPS'2000 Workshop on Model Theoretic Approaches to Planning*, pages 55–61, 2000.
- [GGN⁺04] I. Gent, E. Giunchiglia, M. Narizzano, A. Rowley, and A. Tacchella. Watched data structures for QBF solvers. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, (SAT)*, volume 2919 of *LNCS*, pages 25–36. Springer, 2004.
- [GHR03] I. P. Gent, H. H. Hoos, A. G. D. Rowley, and K. Smyth. Using stochastic local search to solve quantified Boolean formulae. In *Proc. CP*, pages 348–362, 2003.
- [GNT01] E. Giunchiglia, M. Narizzano, and A. Tacchella. Backjumping for quantified Boolean logic satisfiability. In *Proc. IJCAI*, pages 275–281, 2001.
- [GNT02] E. Giunchiglia, M. Narizzano, and A. Tacchella. Learning for Quantified Boolean Logic Satisfiability. In *Proc. 18th National Conference on Artificial Intelligence (AAAI) (AAAI'2002)*, pages 649–654, 2002.
- [GNT03] E. Giunchiglia, M. Narizzano, and A. Tacchella. Backjumping for Quantified Boolean Logic Satisfiability. *Artificial Intelligence*, 145:99–120, 2003.
- [GNT04] E. Giunchiglia, M. Narizzano, and A. Tacchella. Monotone literals and learning in QBF reasoning. In *Tenth International Conference on Principles and Practice of Constraint Programming, CP 2004*, pages 260–273, 2004.
- [GNT06] E. Giunchiglia, M. Narizzano, and A. Tacchella. Clause/term resolution and learning in the evaluation of quantified Boolean formulas. *Journal of Artificial Intelligence Research (JAIR)*, 26:371–416, 2006.
- [GNT07] E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantifiers structure in search based procedures for QBF. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(3):497–507, March 2007.
- [GR04] I. P. Gent and A. G. D. Rowley. Encoding connect-4 using quantified boolean formulae. In *Proc. 2nd International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pages 78–93, February 2004.
- [HBS06] M. Herbstritt, B. Becker, and C. Scholl. Advanced sat-techniques for bounded model checking of blackbox designs. In *Proc. MTV*, pages 37–44, 2006.
- [Hen61] L. A. Henkin. Some remarks on infinitely long formulas. In *Infinite Methods: Proceedings of the Symposium on Foundations of Mathematics*, pages 167–183. Pergamon Press and Państwowe Wydawnictwo Naukowe, 1961.
- [JB07] T. Jussila and A. Biere. Compressing bmc encodings with qbf. *Electr. Notes Theor. Comput. Sci.*, 174(3):45–56, 2007.
- [JS04] P. Jackson and D. Sheridan. Clause form conversions for Boolean

- circuits. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, pages 183–198, May 2004.
- [KBKF95] H. Kleine Büning, M. Karpinski, and A. Flögel. Resolution for quantified Boolean formulas. *Information and Computation*, 117(1):12–18, 1995.
- [KHD05] J. Katz, Z. Hanna, and N. Dershowitz. Space-efficient bounded model checking. In *Proc. DATE*, pages 686–687, 2005.
- [Let02] R. Letz. Lemma and model caching in decision procedures for quantified Boolean formulas. In *Proceedings of Tableaux 2002*, LNAI 2381, pages 160–175. Springer, 2002.
- [LSB05] A. C. Ling, D. P. Singh, and S. D. Brown. FPGA logic synthesis using quantified Boolean satisfiability. In *Proc. SAT*, pages 444–450, 2005.
- [MS04] M. N. Mneimneh and K. A. Sakallah. Computing vertex eccentricity in exponentially large graphs: QBF formulation and solution. In *Theory and Applications of Satisfiability Testing, 6th International Conference, (SAT)*, volume 2919 of *LNCIS*, pages 411–425. Springer, 2004.
- [MSS96] J. P. Marques-Silva and K. A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 220–227, November 1996.
- [MVB07] H. Mangassarian, A. Veneris, and M. Benedetti. Fault diagnosis using quantified Boolean formulas. In *Proc. IEEE Silicon Debug and Diagnosis Workshop (SDD)*, May 2007.
- [OSTW06] J. Oetsch, M. Seidl, H. Tompits, and S. Woltran. ccT: A correspondence-checking tool for logic programs under the answer-set semantics. In *Proc. JELIA*, pages 502–505, 2006.
- [Pap94] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, Mass., 1994.
- [PBZ03] D. A. Plaisted, A. Biere, and Y. Zhu. A satisfiability procedure for quantified Boolean formulae. *Discrete Applied Mathematics*, 130(2):291–328, 2003.
- [PG86] D. A. Plaisted and S. Greenbaum. A Structure-preserving Clause Form Translation. *Journal of Symbolic Computation*, 2:293–304, 1986.
- [PT07] L. Pulina and A. Tacchella. A multi-engine solver for quantified Boolean formulas. In *Proc. CP*, pages 494–497, 2007.
- [PV03] G. Pan and M. Y. Vardi. Optimizing a BDD-based modal solver. In *Proc. CADE-19*, pages 75–89, 2003.
- [PV04] G. Pan and M. Y. Vardi. Symbolic decision procedures for qbf. In *Proc. CP*, pages 453–467, 2004.
- [Rin99a] J. Rintanen. Constructing conditional plans by a theorem prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.
- [Rin99b] J. Rintanen. Improvements to the evaluation of Quantified Boolean Formulae. In D. Thomas, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99-Vol2)*,

- pages 1192–1197, S.F., July 31–August 6 1999. Morgan Kaufmann Publishers.
- [Rob68] A. Robinson. The generalized resolution principle. In *Machine Intelligence*, volume 3, pages 77–93. Oliver and Boyd, Edinburgh, 1968. Reprinted in [SW83].
- [SAG⁺06] A. Sabharwal, C. Ansótegui, C. P. Gomes, J. W. Hart, and B. Selman. QBF modeling: Exploiting player symmetry for simplicity and efficiency. In *Proc. SAT*, pages 382–395, 2006.
- [SB01] C. Scholl and B. Becker. Checking equivalence for partial implementations. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 238–243, 2001.
- [SB05] H. Samulowitz and F. Bacchus. Using SAT in QBF. In *CP*, pages 578–592, 2005.
- [SB06] H. Samulowitz and F. Bacchus. Binary clause reasoning in QBF. In *SAT*, pages 353–367, 2006.
- [SW83] J. Siekmann and G. Wrightson, editors. *Automation of Reasoning: Classical Papers in Computational Logic 1967–1970*, volume 1-2. Springer-Verlag, 1983.
- [Tom03] H. Tompits. Expressing default abduction problems as quantified boolean formulas. *AI Communications*, 16(2):89–105, 2003.
- [Tse70] G. Tseitin. On the complexity of proofs in propositional logics. *Seminars in Mathematics*, 8, 1970. Reprinted in [SW83].
- [Tur02] H. Turner. Polynomial-length planning spans the polynomial hierarchy. In *Proc. JELIA*, pages 111–124, 2002.
- [Zha06] L. Zhang. Solving QBF by combining conjunctive and disjunctive normal forms. In *Proc. AAI*, 2006.
- [ZM02a] L. Zhang and S. Malik. Conflict driven learning in a quantified Boolean satisfiability solver. In *Proceedings of International Conference on Computer Aided Design (ICCAD'02)*, 2002.
- [ZM02b] L. Zhang and S. Malik. Towards a symmetric treatment of satisfaction and conflicts in quantified Boolean formula evaluation. In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming*, pages 200–215, 2002.
- [ZMMM01] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *International Conference on Computer-Aided Design (ICCAD'01)*, pages 279–285, November 2001.

Chapter 31

Quantified Boolean Formulas

Olaf Beyersdorff, Mikoláš Janota, Florian Lonsing, Martina Seidl

31.1. Introduction

Motivated by the success achieved in practical SAT solving, *quantified Boolean formulas* (QBFs), the extension of propositional formulas with existential and universal quantifiers over the propositional variables, have become subject of intensive research (cf. Chapter 29 and Chapter 30). Over the last two decades, many different solving approaches have been presented forming a wide, heterogenous landscape of solving paradigms. While in SAT solving the conflict-driven clause learning (CDCL) is clearly the predominant solving paradigm (cf. Chapter 4), the picture for QBFs is less obvious. Two paradigms together with preprocessing have particularly impacted the progress in QBF research over the last years: search-based QBF solving and expansion-based solving.

From empirical evaluations [LSVG16, JJK⁺14], it has become evident that search-based solvers (which have variants of Q-resolution as underlying proof systems) and expansion-based solvers (which are based on the the $\forall\text{Exp}+\text{Res}$ proof system) often show orthogonal behavior. This observation could be concisely captured by the means of proof theory. In particular, separation and simulation results between the the different variants of Q-resolution and $\forall\text{Exp}+\text{Res}$ give a detailed picture of the relations between the different proof systems. For such proof-theoretical characterizations, however, it turned out that propositional techniques are insufficient for QBFs. To this end, novel separation techniques—for example, based on strategy extraction—have been introduced that provide an elegant tool for comparing proof systems.

In practical QBF solving proofs play another important role. Proofs that are polynomially checkable can be used to certify the correctness of the result of a QBF solver. In addition, they can be used to extract strategies in terms of so-called Skolem and Herbrand functions. For example, in the case of a synthesis problem, such a function represents the required implementation, or in the case of planning such a function represents the plan.

To obtain certification and strategy extraction also if preprocessing is used, the QRAT proof system has been introduced. It generalizes the RAT proof system

for SAT which is currently the formal foundation of the certificates produced by recent SAT solvers (see Chapter 15).

Currently, we see a strong interaction between proof theory and solving. In this chapter, we survey this connection between theory and practice. In Section 31.3 QCDCL and Q-resolution are presented and in Section 31.4 we show how expansion-based solving and $\forall\text{Exp}+\text{Res}$ are related. In Section 31.5 we discuss proofs for recent preprocessing techniques. Strategy extraction from proofs is shortly presented in Section 31.6. Section 31.7 compares different proof systems used in solving and introduces powerful proof systems that are currently not implemented in any solvers.

31.2. Preliminaries

A *literal* l is a Boolean variable x or its negation \bar{x} and $\text{var}(l) = x$. By $\bar{\bar{l}}$, we denote the complementary literal of l , i.e., if $l = x$ then $\bar{\bar{l}} = \bar{x}$ and if $l = \bar{x}$ then $\bar{\bar{l}} = x$. A *clause* is a disjunction of literals (sometimes also interpreted as set of literals) and a *term* is a conjunction of literals. The empty clause is denoted by \perp . A propositional formula in *conjunctive normal form* (CNF) is a conjunction of clauses. We sometimes write a formula as a set of clauses. By $\text{var}(\psi)$ we denote the set of variables that occur in a formula ψ .

The *quantified Boolean formulas* (QBFs) considered in this chapter are in closed prenex conjunctive normal form (PCNF). A QBF $\Pi\psi$ consists of a matrix ψ and a prefix $\Pi = \mathcal{Q}_1X_1 \dots \mathcal{Q}_kX_k$ where $\mathcal{Q}_i \in \{\exists, \forall\}$, $\mathcal{Q}_i \neq \mathcal{Q}_{i+1}$, and X_i are pairwise disjoint sets of variables. Furthermore, $\text{var}(\psi) \subseteq \text{var}(\Pi)$ where $\text{var}(\Pi)$ denotes the union of the X_i . If $x \in X_i$, we say that x is quantified at level i and write $\text{lv}(x) = i$; we write $\text{lv}(l)$ for $\text{lv}(\text{var}(l))$. If $\text{lv}(l) = i$, $\text{quant}(\Pi, l) = \mathcal{Q}_i$. We call \mathcal{Q}_iX_i a *quantifier block*; \mathcal{Q}_1X_1 is the *outermost* quantifier block and \mathcal{Q}_kX_k is the *innermost* quantifier block. For two variables x, y , we have $x \leq_\Pi y$ iff $x \in X_i$ and $y \in X_j$ and $i \leq j$. We use the standard QBF semantics: a QBF $\forall x\Pi\psi$ is true if $\Pi\psi[0/x]$ and $\Pi\psi[1/x]$ are true ($\Pi\psi[t/x]$ denotes the replacement of x by t). Further $\exists x\Pi\psi$ is true if $\Pi\psi[0/x]$ or $\Pi\psi[1/x]$ is true. We also write 1 for true and 0 for false. Often the evaluation of a QBF $\mathcal{Q}_1X_1 \dots \mathcal{Q}_kX_k(\psi)$ is seen as a *game* between the *universal* and the *existential player*. In the i -th step of the game, the player \mathcal{Q}_i assigns values to all the variables X_i . The existential player wins the game iff the matrix ψ evaluates to true under the assignment constructed in the game. The universal player wins iff the matrix ψ evaluates to false. For a universal variable u (an existential variable e), a *strategy* is a Boolean function over the existential (universal) variables that occur to the left of u (e) in the prefix. A QBF is (false) iff there exists a *winning strategy* for the existential (universal) player, i.e. if the existential (universal) player has a strategy for all existential (universal) variables such that this player wins any game when assigning the variable according to this strategy. The Boolean functions forming a winning strategy of the existential (universal) player are often called *Skolem functions* (*Herbrand functions*). Note that given a QBF, a winning strategy always exists for one and only one of the players.

A *proof system* for a language L over an alphabet Γ (usually binary) is a polynomial-time computable partial function $f : \Gamma^* \rightarrow \Gamma^*$ with $\text{rng}(f) = L$. If

$f(x) = y$ then x is called an f -proof for y . If L consists of all propositional tautologies, then f is called a *propositional proof system*, and proof systems for the language TQBF of true QBFs are called *QBF proof systems*. Equivalently, we can consider refutation proof systems where we start with the negation of the formula that we want to prove and derive a contradiction. A proof system S for L *simulates* a proof system P for L if there exists a polynomial p such that for all P -proofs π of x there is an S -proof π' of x with $|\pi'| \leq p(|\pi|)$. If such a proof π' can even be computed from π in polynomial time we say that S *p-simulates* P .

31.3. Proof Systems Based on Q-Resolution

Since the 1960s, the resolution principle [Rob65] has become a state-of-the-art approach in various domains of automated reasoning, cf. [BG01]. In the field of propositional satisfiability testing (SAT), groundbreaking technology to leverage the power of resolution in CDCL SAT solvers was developed in the late 1990s [MSS99] and early 2000s [MMZ⁺01, ZMMM01], cf. Chapter 4. Almost 20 years later, resolution still is the basic paradigm that underlies modern SAT solvers, which selectively integrate resolution with more powerful proof systems to potentially generate shorter proofs [WHHJ14, HKB17].

The resolution principle has been lifted to the QBF setting [KBKF95], along with the CDCL approach called QCDCL [GNT06, Let02, ZM02a]. In the following, we present the *Q-resolution calculus* and variants of it as the formal foundation of QCDCL. Additionally, we point out how the workflow of QCDCL interacts with the rules of the Q-resolution calculus.

31.3.1. The Q-Resolution Calculus

Resolution is one of the best studied propositional proof systems (cf. Chapter 7). It is a refutational proof system manipulating unsatisfiable CNFs as sets of clauses. The only inference rule *propres* is defined by

$$\frac{C \cup \{p\} \quad D \cup \{\bar{p}\}}{C \cup D} \quad (\text{propres})$$

where C, D are clauses and p is a literal. A *resolution refutation* derives the empty clause \emptyset . The generalization to QBF is shown in Figure 31.1.

We refer to a clause containing complementary literals as being *tautological* or a *tautology*. Non-tautological clauses occurring in the CNF of the given QBF ϕ are selected by applications of the axiom rule *cl-init*. The resolution rule *res* is similar to the resolution rule in propositional logic. However, the *pivot variable* p is restricted to existential variables and derived clauses must not be tautological.

The main distinguishing feature between propositional resolution and Q-resolution (Q-Res) is the rule *red*, the *reduction* operation. *Universal reduction* eliminates *trailing* universal literals locally from a non-tautological clause C with respect to the linear quantifier ordering. A universal literal l is trailing in a clause C if all existential literals $l' \in C$ are smaller than l . Note that universal reduction is the only rule in the calculus that allows to eliminate universal literals from

$\frac{C \cup \{l\}}{C} \quad \text{for all } x \in \text{vars}(\Pi) : \{x, \bar{x}\} \not\subseteq (C \cup \{l\}), \text{ quant}(\Pi, l) = \forall, \text{ and } l' <_{\Pi} l \text{ for all } l' \in C \text{ with } \text{quant}(\Pi, l') = \exists$	(red)
$\frac{C_1 \cup \{p\} \quad C_2 \cup \{\bar{p}\}}{C_1 \cup C_2} \quad \text{for all } x \in \text{vars}(\Pi) : \{x, \bar{x}\} \not\subseteq (C_1 \cup C_2), \bar{p} \notin C_1, p \notin C_2, \text{ and } \text{quant}(\Pi, p) = \exists$	(res)
$\frac{}{C} \quad \text{for all } x \in \text{vars}(\Pi) : \{x, \bar{x}\} \not\subseteq C \text{ and } C \in \psi$	(cl-init)

Figure 31.1. The rules of the Q-Resolution Calculus (Q-Res) [KBKF95]: $\phi = \Pi\psi$ is a PCNF, C a clause, and x, l and p are literals.

clauses. As such, it is crucial to obtain the empty clause in refutations of PCNFs. In a more general setting, it was shown that a QBF proof system can be obtained [BBC16] by combining the universal reduction rule with propositional proof systems.

Example 1. Consider the PCNF $\phi = \exists x_1, x_3, x_4 \forall y_5 \exists x_2. (\bar{x}_1 \vee x_2) \wedge (x_3 \vee y_5 \vee \bar{x}_2) \wedge (x_4 \vee \bar{y}_5 \vee \bar{x}_2) \wedge (\bar{x}_3 \vee \bar{x}_4)$. We resolve $(\bar{x}_3 \vee \bar{x}_4)$ and $(x_4 \vee \bar{y}_5 \vee \bar{x}_2)$ on the existential variable x_4 to obtain $C_0 := (\bar{x}_3 \vee \bar{y}_5 \vee \bar{x}_2)$. Note that the universal literal \bar{y}_5 cannot be reduced by universal reduction due to literal \bar{x}_2 . We resolve $C_0 = (\bar{x}_3 \vee \bar{y}_5 \vee \bar{x}_2)$ and $(\bar{x}_1 \vee x_2)$ on x_2 and get $C_1 := (\bar{x}_1 \vee \bar{x}_3 \vee \bar{y}_5)$. Now literal \bar{y}_5 is trailing in C_1 and can be reduced, which produces the clause $C_2 := (\bar{x}_1 \vee \bar{x}_3)$.

The above rules in Figure 31.1 define the common variant of Q-resolution, which is sound and refutational-complete for QBFs in PCNF [KBKF95]. The advantage of imposing restrictions in the side conditions of the rules of the calculus is that the search for a possible proof is made easier because the number of possible proof steps is reduced. However, lifting these restrictions, either individually or in combined ways, results in more powerful variants of Q-resolution.

In the following, we discuss variants of Q-resolution that result from lifting restrictions or from defining rules that provide additional reasoning capabilities.

31.3.2. Extensions of Q-Resolution

Long-Distance Q-resolution. The CDCL paradigm for SAT solving was lifted to the QBF level soon after its inception [GNT06, Let02, ZM02a]. When deriving a learned clause C in QCDCL analogously to CDCL following the *first unique implication point (1-UIP)* principle, C might end up being a tautology caused by a pair of complementary universal literals [ZM02a]. Resolution steps producing tautologies are called *long-distance resolution* steps because the tautology is produced by resolving two clauses that contain two literals that appear positively in one clause and negatively in the other. In general, deriving and using tautological clauses as part of refutations is unsound, as the following example shows.

Example 2. Consider the satisfiable PCNF $\phi := \forall x \exists y. (x \vee \bar{y}) \wedge (\bar{x} \vee y)$ which is obviously satisfiable. In an attempt to produce a *wrong* refutation, we resolve

$\frac{C \cup \{l\}}{C}$	$l \in \{y, \bar{y}, y^*\}, \text{quant}(\Pi, l) = \forall, \text{ and}$ $l' <_{\Pi} l \text{ for all } l' \in C \text{ with } \text{quant}(\Pi, l') = \exists$	$(ldqres)$
$\frac{C_1 \cup \{p\} \cup X_1 \quad C_2 \cup \{\bar{p}\} \cup X_2}{C_1 \cup C_2 \cup X^*}$	$\bar{p} \notin C_1, p \notin C_2, \text{quant}(\Pi, p) = \exists, \text{ and}$ $p <_{\Pi} l \text{ for all } l \in X_2$	$(ldqres)$

Figure 31.2. The long-distance Q-resolution calculus (LD-Q-Res) [ZM02a, BJ12] consists of the rules shown above in addition to the axiom rule *cl-init* from Figure 31.1. Let $\phi = \Pi\psi$ be a formula in PCNF, C a clause, and l and p literals. The sets X_1, X_2 , and X^* in rule *ldqres* contain (merged or unmerged) universal literals such that the following conditions hold. For each $y \in X_1$, if y is not a merged literal, then either $\bar{y} \in X_2$ or $y^* \in X_2$, and otherwise either $y \in X_2, \bar{y} \in X_2$ or $y^* \in X_2$. The set X^* contains the merged literal of each literal in X_1 .

the two clauses on variable y by neglecting the side condition of rule *res*, which produces the clause $(x \vee \bar{x})$. From that clause, we derive the empty clause by universal reduction.

In contrast to the observation made in Example 2, tautologies derived during learning in QCDCL can be used in the same way as ordinary, non-tautological learned clauses to prune the search space. In related work [BJ12], long-distance resolution (LD-Q-Res) was formulated as a calculus shown in Figure 31.2. In the following, we use the common notation y^* to denote the occurrence of the pair $y \vee \bar{y}$ of complementary universal literals in a clause. This special literal y^* is called a *merged literal*.

Tautologies resulting from resolution steps according to rule *ldqres* are always caused by complementary universal literals and never by existential ones. Furthermore, for the soundness of the calculus, it is necessary to respect the quantification level restriction of the pivot variable and the merged universal literals. Under that restriction, the wrong refutation in Example 2 cannot be produced. Furthermore, the QCDCL framework by construction guarantees that any tautologies derived during learning are derived under that restriction. Universal reduction by rule *ldqred* also allows the elimination of merged literals. The ability to produce certain tautological resolvents makes LD-Q-Res stronger than Q-Res [ELW13], cf. Section 31.7.

Example 3. Consider the PCNF $\phi = \exists x_1, x_3, x_4 \forall y_5 \exists x_2. (\bar{x}_1 \vee x_2) \wedge (x_3 \vee y_5 \vee \bar{x}_2) \wedge (x_4 \vee \bar{y}_5 \vee \bar{x}_2) \wedge (\bar{x}_3 \vee \bar{x}_4)$ and the first resolvent $C_0 := (\bar{x}_3 \vee \bar{y}_5 \vee \bar{x}_2)$ from Example 1. We carry out a long-distance resolution step on C_0 using the clause $(x_3 \vee y_5 \vee \bar{x}_2)$ and the pivot variable x_3 , which results in $C_1 := (y_5^* \vee \bar{x}_2)$. Note that the pivot variable x_3 has a smaller quantification level than the literals y_5 and \bar{y}_5 that result in the merged literal y_5^* according to rule *ldqres*.

Universal Pivots, Long-Distance Q-Resolution, and Combinations. The restriction of the quantifier type of pivot variables in Q-Res shown in Figure 31.1 can be relaxed to allow both existential and universal pivots in resolution steps. The resulting variant is called QU-resolution (QU-Res) and was shown to be stronger than Q-Res [VG12a]. Despite the possibility to eliminate universal literals

from clauses by resolving on them when constructing a resolution refutation, the universal reduction rule is still crucial for the completeness of QU-Res.

Further variants of Q-resolution were obtained by combining the relaxation of the quantifier type restriction of pivot variables and the generation of certain tautological resolvents [BWJ14]. LQU-resolution extends long-distance Q-resolution by also admitting universal pivot variables like in QU-resolution. However, resolution steps on universal pivots must not produce tautological resolvents. This restriction is relaxed in LQU⁺-resolution (LQU⁺-Res), which is thus a generalization of LQU-resolution.

Regarding proof complexity, both QU-Res and LD-Q-Res are stronger than Q-Res while being incomparable [BWJ14] to each other. LQU⁺-Res is stronger than both QU-Res and LD-Q-Res (cf. Section 31.7).

Enhanced Universal Reduction: Dependency Schemes. The linear ordering of the quantifier prefix of PCNFs has to be taken into account in practical QBF solving as well as in the theoretical context of QBF calculi, e.g., in the side condition of the universal reduction rule *red* in the Q-resolution calculus. The quantifier ordering imposes restrictions as it both limits the freedom of a QBF solver to assign variables and the potential number of universal literals eliminated by universal reduction. As a way to overcome these restrictions, *dependency schemes* [Sam08, SS09] were introduced to relax the linear quantifier ordering.

Informally, a dependency scheme is a binary relation D over the set of the variables of a QBF that expresses variable *independence*. Given two variables x and y , if $(x, y) \notin D$ then a QBF solver may safely assume that the values of x and y can be chosen independently from each other. Otherwise, if $(x, y) \in D$, then it has to assume that the value of one depends on the value of the other. Neglecting the dependency of the values of variables, e.g., bears the risk of unsound results in solving.

Dependency schemes can be regarded as an approximate representation of the *actual* independence of variables. For two variables, actual independence means that their values can be chosen independently from each other without affecting the result of a semantical evaluation of a QBF. Checking whether two variables in a QBF are independent is as hard as solving the QBF itself, i.e., PSPACE-complete. Therefore, dependency schemes represent sound approximations of the actual independence of variables in a QBF. These approximations result in the presence of *spurious dependency pairs*. A dependency scheme D may contain spurious dependency pairs (x, y) such that $(x, y) \in D$ but still the values of x and y can be chosen independently. A dependency scheme D is considered to be more refined than some other scheme D' if D contains fewer spurious dependency pairs than D' . The potential amount of spurious dependency pairs present in a dependency scheme is expressed by a hierarchy of schemes defined by different levels of refinement. For practical applications, only dependency schemes are of interest that can be computed in polynomial time such as the *standard dependency scheme* [SS09] or the resolution path dependency scheme [VG11, SS12], the latter being the most refined, practically relevant scheme currently known. In contrast to that, the *trivial dependency scheme* exactly represents the linear ordering of the quantifier prefix and thus is least refined.

With respect to Q-resolution calculi, dependency schemes are relevant in that they can be used to strengthen the universal reduction rule as follows.

Definition 1 (Dependency-Aware Reduction [LB10]). Let $\phi = \Pi\psi$ be a PCNF and D be a dependency relation represented by a dependency scheme.

$$\frac{C \cup \{l\}}{C} \quad C \text{ is a clause, } \text{quant}(\Pi, l) = \forall, \text{ and,} \quad (dep\text{-red}) \\ (l, l') \notin D \text{ for all } l' \in C \text{ with } \text{quant}(\Pi, l') = \exists$$

Dependency-aware universal reduction by rule *dep-red* generalizes the traditional universal reduction rule *red* of the Q-resolution calculus (Figure 31.1). Instead of the linear ordering of variables (\leq_{Π}) in the prefix of the PCNF ϕ , rule *dep-red* makes use of a dependency relation D . A universal literal l can be eliminated from a clause C if the variable of l is independent from the variables of all existential literals l' in C .

Note that rule *dep-red* indeed generalizes traditional universal reduction by rule *red* since it can be instantiated by the trivial dependency scheme. With dependency-aware universal reduction, it is possible to potentially reduce universal literals that cannot be reduced by traditional universal reduction, depending on the actual dependency relation used in rule *dep-red*.

Q-resolution calculi can be equipped with the dependency-aware universal reduction rule based on tractable dependency schemes. Using the reflexive resolution path scheme, e.g., a variant of Q-resolution is obtained that is stronger than traditional Q-resolution [BB17] (cf. Section 31.7.5). Certain theoretical properties of dependency schemes were studied that are sufficient to prove the soundness of combining these schemes with universal reduction in Q-resolution calculi, including long-distance Q-resolution [BB16, PSS16, SS16, PSS18].

The benefits of exploiting variable independence were also empirically observed in the context of solving. Early approaches are based on exploiting quantifier structure that is present in the parse tree of a given QBF [GNT07]. Alternatively, such structure can be (partially) reconstructed from a PCNF by shifting quantifiers inside the CNF part [AB02, Ben05], which is also called *mini-scoping*. For example, the expansion-based solver Quantor [Bie04] applies a limited form of mini-scoping to allow for a better scheduling of expansion steps. Dependency schemes can be regarded as a generalization of mini-scoping. Already the standard dependency scheme [SS09], which is less refined than the resolution path scheme [VG11, SS12], potentially allows to identify more independent variables than mini-scoping [LB09].

Since QCDCL solvers rely on Q-resolution, dependency-aware universal reduction can be applied to leverage variable independence in QCDCL. The QCDCL solver DepQBF [LE17] tightly integrates dependency-aware universal reduction via the standard dependency scheme in the QCDCL workflow [LB10]. Additionally, it employs variable independence to generate assignments in QCDCL on a more flexible basis than in traditional QCDCL, which operates with the linear quantifier order. The approach implemented in DepQBF was generalized in the QCDCL solver Qute [PSS17], which starts with an empty dependency relation and learns dependency pairs on demand.

Q-Resolution with Symmetries. Another powerful extension of the basic Q-Resolution Calculus (cf. Figure 31.1) exploits symmetries [KS18b] of a given formula. A symmetry for a QBF $\Pi\psi$ is a bijective map $\sigma: L \rightarrow L$ where $L = \{x, \bar{x} \mid x \in \text{var}(\Pi)\}$ such that for all $x \in \text{var}(\Pi)$ $\sigma(\bar{x}) = \overline{\sigma(x)}$ and $x, \sigma(x)$ belong to the same quantifier block. Further, if σ is applied to all literals in ψ , then ψ maps to itself.

Definition 2 (Symmetry Rule [KS18a]). Let $\phi = \Pi\psi$ be a PCNF.

$$\frac{C}{\sigma(C)} \quad C \text{ is a clause, } \sigma \text{ is a symmetry of } \Pi\psi \quad (\text{sym})$$

With the symmetry rule formula families hard for Q-Res such as $QParity_n$ or $KBKF_n$ have polynomial-size proofs [KS18a] (cf. Section 31.7.5).

31.3.3. Solving with Q-Resolution: QCDCL

Conflict-driven clause learning, called QCDCL [GNT06, Let02, ZM02a, ZM02b], is a generalization of the CDCL approach for SAT solving, cf. Chapter 4. A QCDCL solver derives new learned clauses during the search with the purpose to prune the search space. Depending on the variable assignments that are successively generated, the solver may also produce *learned cubes*. Dual to clauses, a cube is a conjunction of literals. While learned clauses prune parts of the search space that contain falsifying assignments, learned cubes prune parts that contain satisfying assignments. Propositional resolution and Q-resolution [KBKF95] are the proof systems that underlie CDCL and QCDCL. For unsatisfiable and satisfiable QBFs, a QCDCL solver can produce clause and cube proofs, respectively. Cube proofs are formulated in a calculus that is dual to Q-resolution of clauses.

Despite the similarity between CDCL and QCDCL on a theoretical level, clause learning in a QCDCL solver is more complex than in a CDCL solver. This is due to the interaction between the QCDCL workflow and the rules of the Q-resolution calculus. In the following, we point out this interaction on an abstract level. In particular, we do not give a comprehensive presentation of QCDCL. Instead, we refer to Chapter 30, where clause learning based on Q-resolution is presented in the framework of DPLL-based backtracking search, called *Q-DLL* or, alternatively, *QDPLL* [CGS98a, GNT06]. In terms of the effects of clause learning, QDPLL and QCDCL are similar. However, we find the QCDCL framework more suitable for our purposes due to its immediate relation to CDCL. Moreover, for simplicity we focus on clause learning and hence omit cube learning in our presentation of QCDCL.

Figure 31.3 shows a high-level flowchart of the QCDCL workflow. The building blocks are similar to CDCL: propagation in stage QBCP (quantified Boolean constraint propagation), decision making, learning, and backtracking. Given a PCNF $\phi = \Pi\psi$, variable assignments are successively enumerated by means of QBCP and decision making. The formula is simplified under the current assignment A to obtain the formula ϕ' . If ϕ' is not reduced to a syntactic truth constant \perp or \top then the current assignment is further extended by decision making and successive application of QBCP.

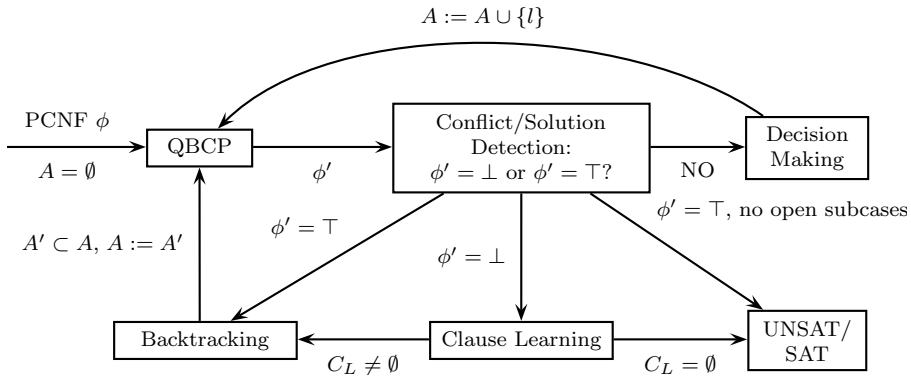


Figure 31.3. Flowchart of QCDCL. For simplicity of the presentation, cube learning is not shown. Instead, we assume that backtracking is carried out immediately after a satisfiable subcase has been detected ($\psi' = \top$).

Otherwise, if $\phi' = \top$ then a satisfiable subcase has been determined and closed. In this case, if there are no more open subcases remaining to be explored, then QCDCL terminates with the result that the input formula ϕ is satisfiable. If there are open subcases remaining, then a certain part A' of the current assignment A is retracted during backtracking, and a new assignment A is generated. In implementations of QCDCL, determining whether there are open subcases left and backtracking after finding satisfiable subcases is driven by cube learning, which we omit in our presentation. Instead, we assume that backtracking is carried out immediately after a satisfiable subcase has been detected.

Case $\phi' = \perp$ designates an unsatisfiable subcase, which triggers clause learning. A new learned clause C_L is derived by Q-resolution using clauses depending on assignment A . If C_L is empty ($C_L = \emptyset$), then QCDCL terminates with the result that the input formula ϕ is unsatisfiable. In this case, a Q-resolution proof of ϕ can be obtained by considering the derivations of the learned clauses. Otherwise, if C_L is non-empty ($C_L \neq \emptyset$) then it is added to formula ϕ , backtracking is driven depending on C_L , and a new assignment A is generated.

In the following, we describe the interaction of universal reduction and Q-resolution with the workflow of QCDCL. The universal reduction rule *red* of the Q-resolution calculus (Figure 31.1) is crucial not only for the derivation of clauses, but also in the generation of assignments by propagation in QBCP. Similar to BCP in CDCL, QBCP in QCDCL consists of unit literal detection. In addition to BCP, QBCP also applies universal reduction to shorten clauses under assignment A that is currently being generated. The combination of unit literal detection and universal reduction potentially detects more assignments and shortens more clauses than unit literal detection alone. In the following, we represent a variable assignment A as a set of literals such that $v \in A$ if variable v is assigned true, and $\neg v \in A$ if v is assigned false, and $v \notin A$ and $\neg v \notin A$ if v is unassigned. For a PCNF ϕ and an assignment A , $\phi[A]$ is the formula simplified under A .

Definition 3 (Unit Literal Detection [CGS98b]).

Given a QBF ϕ , a clause $C \in \phi$ is *unit* iff $C = (l)$ and $q(l) = \exists$. The existential literal l in C is called a *unit literal*. *Unit literal detection (UL)* collects the assignment $\{l\}$ from the unit clause $C = (l)$. Unit literal detection applied to a QBF ϕ collects the respective assignments from all unit clauses in ϕ .

Learned clauses C_L generated in QCDCL have the property that they become unit after backtracking. Similar to clause learning in CDCL, such clauses are *asserting* clauses. Note that the variables of unit literals are restricted to existential ones. This is necessary for the combination of unit literal detection and universal reduction in QBCP.

Definition 4 (QBCP). Given a PCNF ϕ and the empty assignment $A = \{\}$, i.e. $\psi[A] = \phi$.

1. Apply universal reduction (UR) to $\phi[A]$ to obtain $\phi[A]'$.
2. Apply unit literal detection (UL) to $\phi[A]'$ to collect new assignments.
3. Add assignments found by UL to A and repeat steps 1-2.
4. Stop if A does not change anymore or if $\phi[A] = \top$ or $\phi[A] = \perp$.

The result of QBCP applied to ϕ is an extended assignment A and a simplified formula $\phi[A]'$ obtained from $\phi[A]$ by UR.

Pure literal detection [CGS98b] is another approach to generate assignments in QBCP in addition to unit literal detection. It can further increase the capability of QBCP to generate assignments and shorten clauses. For simplicity of the presentation, however, we do not present pure literal detection.

Example 4. Let ϕ be a PCNF as follows:

$$\forall y_5 \exists x_1 \forall y_2 \exists x_3, x_4. (\bar{y}_5 \vee x_4) \wedge (y_5 \vee \bar{x}_4) \wedge (x_1 \vee y_2 \vee \bar{x}_4) \wedge (\bar{x}_1 \vee x_3 \vee \bar{x}_4) \wedge (\bar{y}_2 \vee \bar{x}_3)$$

Initially, no simplifications of ϕ by QBCP are possible. We assign y_5 by decision making to obtain assignment $A = \{y_5\}$. The formula simplifies to

$$\phi[y_5] = \exists x_1 \forall y_2 \exists x_3, x_4. (x_4) \wedge (x_1 \vee y_2 \vee \bar{x}_4) \wedge (\bar{x}_1 \vee x_3 \vee \bar{x}_4) \wedge (\bar{y}_2 \vee \bar{x}_3)$$

By UL and UR, we obtain the following assignments and simplified formulas.

- By UL: $\phi[y_5, x_4] = \exists x_1 \forall y_2 \exists x_3. (x_1 \vee y_2) \wedge (\bar{x}_1 \vee x_3) \wedge (\bar{y}_2 \vee \bar{x}_3)$.
- By UR: $\phi[y_5, x_4] = \exists x_1 \forall y_2 \exists x_3. (x_1) \wedge (\bar{x}_1 \vee x_3) \wedge (\bar{y}_2 \vee \bar{x}_3)$.
- By UL: $\phi[y_5, x_4, x_1] = \forall y_2 \exists x_3. (x_3) \wedge (\bar{y}_2 \vee \bar{x}_3)$.
- By UL: $\phi[y_5, x_4, x_1, x_3] = \forall y_2. (\bar{y}_2)$.
- By UR: $\phi[y_5, x_4, x_1, x_3] = \emptyset$

Since a conflict (i.e., empty clause) is derived in the last step, by QBCP we have shown that $\phi[y_5]$ and $\phi[y_5, x_4, x_1, x_3]$ are satisfiability-equivalent, i.e., $\phi[y_5]$ is unsatisfiable. Since the decision variable y_5 is universal, also ϕ is unsatisfiable. Note that in this example, universal reduction is crucial to determine the unsatisfiability of ϕ because unit literal detection alone cannot derive a conflict.

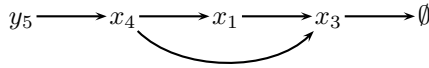


Figure 31.4. Implication graph related to the application of QBCP in Example 4. Variable y_5 is assigned as a decision.

For every assignment l collected by unit literal detection in QBCP the corresponding *antecedent clause* is recorded. The antecedent clause of assignment l is the clause of the original PCNF ϕ that became unit in QBCP. Given a conflict as in Example 4 and the set of antecedent clauses of unit literals in the current assignment, new learned clauses can be derived in QCDCL in a similar but more complex way as in CDCL. We illustrate the derivation of clauses from antecedent clauses in the following example.

Example 5. Let ϕ be the PCNF from Example 4:

$$\forall y_5 \exists x_1 \forall y_2 \exists x_3, x_4. (\bar{y}_5 \vee x_4) \wedge (y_5 \vee \bar{x}_4) \wedge (x_1 \vee y_2 \vee \bar{x}_4) \wedge (\bar{x}_1 \vee x_3 \vee \bar{x}_4) \wedge (\bar{y}_2 \vee \bar{x}_3)$$

The assignment generated by decision making and QBCP in Example 4 is $A := \{y_5, x_4, x_1, x_3\}$, with variables assigned in that ordering. The antecedent clauses of the assignments $\{x_4\}$, $\{x_1\}$, and $\{x_3\}$ are $C_{x_4} := (\bar{y}_5 \vee x_4)$, $C_{x_1} := (x_1 \vee y_2 \vee \bar{x}_4)$, and $C_{x_3} := (\bar{x}_1 \vee x_3 \vee \bar{x}_4)$, respectively. We resolve C_{x_1} and C_{x_4} and obtain $C_1 := (\bar{y}_5 \vee x_1)$ after universal reduction. Further, we resolve C_1 with C_{x_3} and obtain $C_2 := (\bar{y}_5 \vee x_3 \vee \bar{x}_4)$, which we resolve with C_{x_4} to get $C_3 := (\bar{y}_5 \vee x_3)$. Finally, by resolving the clause $(\bar{y}_2 \vee \bar{x}_3)$ that became empty during QBCP in Example 4 and C_3 , we obtain the empty clause by universal reduction.

In Example 5, we constructed a Q-resolution proof of the given PCNF by making what appears as an *ad-hoc* selection of clauses to be resolved. QCDCL solvers use information gathered in QBCP in a *systematic* way to carry out resolution steps to derive learned clauses. Pivot selection in QCDCL is crucial as it determines whether a Q-resolution or a long-distance Q-resolution derivation of the learned clause is obtained. In practical terms, a wrong pivot selection may result in failure to derive a learned clause.

In the following, we illustrate how the pivot selection influences the derivation of a learned clause in QCDCL. To this end, we informally introduce the concept of *implication graphs*.

Example 6. An implication graph related to the application of QBCP in Example 4 is shown in Figure 31.4. The implication graph contains a vertex for every decision variable and variable assigned by unit literal detection. It also contains a special vertex representing the empty clause. We add an edge (x, y) if y was assigned by unit literal detection and \bar{x} appears in the antecedent clause of y . For example, $C_{x_3} := (\bar{x}_1 \vee x_3 \vee \bar{x}_4)$ is the antecedent clause of assignment $\{x_3\}$ and the edges (x_1, x_3) and (x_4, x_3) indicate that \bar{x}_1 and \bar{x}_4 appear in C_{x_3} .

Implication graphs generated in QBCP are very similar to the ones generated by BCP in CDCL. However, universal literals eliminated by universal reduction

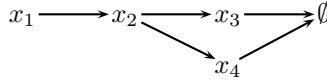


Figure 31.5. Implication graph related to the application of QBCP in Example 7. Variable x_1 is assigned as a decision.

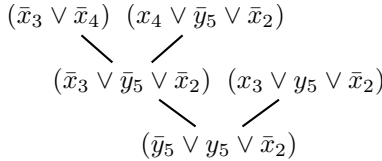


Figure 31.6. Derivation related to Example 7.

are not explicitly represented by an edge. Hence, for example, the clause $(\bar{y}_2 \vee \bar{x}_3)$ that becomes empty in Example 4 has two literals but in the implication graph in Figure 31.4 there is only one edge from x_3 to the special vertex \emptyset . We illustrate the selection of pivot variables and its importance for the derivation of learned clauses by the following example.

Example 7. Let ϕ be a PCNF as follows:

$$\exists x_1, x_3, x_4 \forall y_5 \exists x_2. (\bar{x}_1 \vee x_2) \wedge (x_3 \vee y_5 \vee \bar{x}_2) \wedge (x_4 \vee \bar{y}_5 \vee \bar{x}_2) \wedge (\bar{x}_3 \vee \bar{x}_4)$$

Suppose that we assign x_1 to true as a decision variable. Then we obtain the assignment $A := \{x_1, x_2, x_3, x_4\}$, in that ordering, by QBCP. Figure 31.5 shows the corresponding implication graph. The antecedent clauses of assignments $\{x_2\}$, $\{x_3\}$, and $\{x_4\}$ are $C_{x_2} := (\bar{x}_1 \vee x_2)$, $C_{x_3} := (x_3 \vee y_5 \vee \bar{x}_2)$, and $C_{x_4} := (x_4 \vee \bar{y}_5 \vee \bar{x}_2)$, respectively, and $C_\emptyset := (\bar{x}_3 \vee \bar{x}_4)$ is the clause that becomes empty in QBCP.

Traditional QCDCL solvers derive learned clauses using Q-resolution (Figure 31.1). To this end, pivot variables are selected in reverse assignment ordering. This approach is similar to the pivot selection in CDCL SAT solvers. In our example, we first resolve the falsified clause C_\emptyset with C_{x_4} , which was assigned last, and the resulting clause with C_{x_3} . However, this results in a tautology, as shown in Figure 31.6.

To obtain a legal Q-resolution derivation of a learned clause, tautologies must be avoided. The reason for the tautology produced in Example 7 is that complementary universal literals appearing in relevant antecedent clauses were eliminated by universal reduction in QBCP. To avoid tautologies, such universal literals must be eliminated by universal reduction during the derivation of the learned clause. This can be achieved by selecting existential literals as pivots that block the universal literals from being eliminated. To this end, pivots must be selected in a more flexible way, i.e., not in strict reverse assignment ordering.

Example 8. Referring to Example 7 and Figure 31.6, the tautology can be avoided by first resolving the intermediate clause $(\bar{x}_3 \vee \bar{y}_5 \vee \bar{x}_2)$ with C_{x_2} on

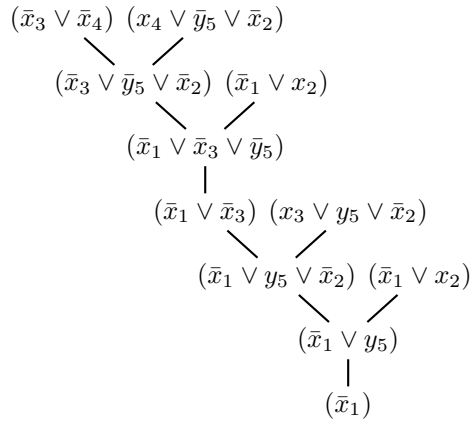


Figure 31.7. Derivation related to Example 8. Nodes having one parent represent the result of universal reduction steps.

variable x_2 to obtain $(\bar{x}_1 \vee \bar{x}_3)$ after eliminating \bar{y}_5 by universal reduction. Note that the pivot variable x_2 is larger than the variable of the universal literal \bar{y}_5 in the quantifier ordering. The clause $(\bar{x}_1 \vee \bar{x}_3)$ is further resolved with C_{x_3} , resulting in $(\bar{x}_1 \vee y_5 \vee \bar{x}_2)$. A final resolution step with C_{x_2} on variable x_2 produces the learned clause (x_1) . The complete Q-resolution derivation is shown in Figure 31.7.

The approach illustrated in Example 8 guarantees that tautologies in the derivation of learned clauses are always avoided [GNT06]. Note that pivot variables were selected in the ordering x_4, x_2, x_3, x_2 , which is in contrast to the reverse assignment ordering x_4, x_3, x_2 . However, this approach has an exponential worst case [VG12a]. The size of the derivation of a single learned clause may be exponential in the size of the implication graph because universal literals eliminated in parts of the derivation may later be reintroduced.

As a remedy to the exponential worst case, *long-distance Q-resolution derivations* of learned clauses can be produced. With this approach, pivots can always be selected in reverse assignment ordering [ELW13], which is similar to the pivot variables selection in CDCL SAT solvers. Moreover, the size of the long-distance Q-resolution derivation of a learned clause is linear in the size of the respective implication graph. Quaffle [ZM02a] was among the first solvers that applied long-distance Q-resolution for clause learning. Figure 31.8 shows a long-distance Q-resolution derivation of the learned clause (x_1) related to Examples 7, and 8.

As an alternative to long-distance Q-resolution derivations, traditional Q-resolution derivations of learned clauses can be generated based on QPUP (QBF pseudo unit propagation) clauses [LEVG13]. In this approach, resolution steps are carried out starting from certain points in the implication graph, and the resolution process proceeds towards the falsified clause. This is in contrast to traditional QCDCL and CDCL SAT solvers, where the resolution process starts at the falsified clause and proceeds towards a unique implication point. In Example 5, we actually constructed a refutation following the QPUP approach (see also the related implication graph in Figure 31.4). QPUP-based learning allows

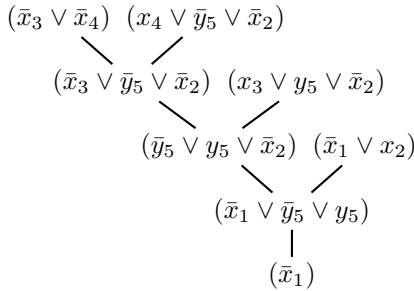


Figure 31.8. Long-distance Q-resolution derivation of the learned clause (x_1) related to Examples 7, and 8. The derivation continues the one shown in Figure 31.6.

to generate Q-resolution derivations that are linear in the size of the implication graph.

We presented several ways to enhance the traditional Q-resolution calculus (Figure 31.1). These include stronger Q-resolution variants, i.e., variants of rule *res*, and stronger variants of universal reduction (rule *red*) based on dependency schemes. The Q-resolution calculus can also be strengthened by a more general variant of the axiom rule *cl-init*. This general variant employs assignments generated in QCDCL and calls to a QBF oracle.

Definition 5 (Generalized Axiom [LBB⁺15, LES16]). Let $\phi = \Pi\psi$ be a PCNF. The *generalized axiom* is defined as follows.

$$\frac{A \text{ is a (partial) assignment generated in QCDCL,} \quad \phi[A] \text{ is unsatisfiable,} \quad (\textit{gen-cl-init})}{C \quad \text{and } C = (\bigvee_{l \in A} \bar{l}) \text{ is a clause}$$

The generalized axiom *gen-cl-init* can be added to the Q-resolution calculus in addition to the traditional axiom *cl-init* in Figure 31.1. Furthermore, it can be combined with any variants of the Q-resolution and universal reduction rules. Applications of the axiom rule require to check the satisfiability of $\phi[A]$. Since this problem is PSPACE-complete in general, for practical applications these satisfiability checks must be bounded and hence are incomplete. If $\phi[A]$ is found unsatisfiable, then the clause C can be derived and used as an ordinary learned clause in QCDCL. Note that learning C is sound only if A is an assignment generated in QCDCL.

Any QBF decision procedure can be used as an oracle to carry out the satisfiability test in the side condition of axiom *gen-cl-init*. Due to this property, the general axiom acts as an interface to integrate arbitrary decision procedures, and hence proof systems, in QCDCL. This way, it is possible to benefit from the combination of proof systems such as Q-resolution and, e.g., bounded expansion by means of clause learning. The resulting variant of QCDCL has the potential to produce proofs that are exponentially shorter than the proofs produced by traditional QCDCL.

Similar to the clause-based variant in Definition 5, a dual, cube-based variant can be formulated [LBB⁺15] for enhanced cube learning.

31.4. Expansion-Based Proof Systems

Taking a different approach than QCDCL, a number of solvers reduce QBF to SAT. Most commonly, they rely on iterative elimination of quantified variables until only one type of quantifier is left. Once the formula contains a single type of quantifier, an off-the-shelf SAT solver is invoked.

One of the first solvers taking this approach is the solver QUBOS [AB02], presented in 2002, which eliminates quantifiers inside-out. Considering some quantifier prefix Π , a QBF $\Pi\exists x\phi$ rewrites to $\Pi(\phi[0/x] \vee \phi[1/x])$ and $\Pi\forall x\phi$ to $\Pi(\phi[0/x] \wedge \phi[1/x])$. We say that the quantifier is *expanded*.

A similar approach is taken in the solver Quantor [Bie04] with the difference that the solver is designed to work on PCNF. That is not straightforward since a disjunction of CNF is no longer CNF. Quantor, in order to preserve PCNF, eliminates existential variables differently from the universal ones. Similarly to the Davis-Putnam procedure (also known as variable elimination, see Chapter 9), existentially quantified variables are exhaustively resolved out of the formula. The solver QMRES [PV04] follows a similar approach but tries to mitigate the size of the expansion by representing sets of clauses as ZBDDs.

A different take on elimination is adopted in the solver sKizzo [Ben04], which operates on a PCNF input and eliminates only universal variables. This, however, requires introduction of fresh existential variables. The reason for such is that if the variable x is eliminated from a formula $\forall x\exists y\dots\phi$, the result is no longer prenex: $(\exists y\dots\phi[0/x])\wedge(\exists y\dots\phi[1/x])$. To get back to prenex form, fresh variables are introduced: $\exists y^0y^1\dots(\phi[0/x,y^0/y] \wedge \phi[1/x,y^1/y])$. In sKizzo the process of introducing fresh variables is referred to as propositional Skolemization because effectively the fresh variables correspond to the individual points of a Skolem function for the original existential variable.

Solvers relying on full expansion of quantifiers inherently suffer from large memory consumption. Some solvers counter this issue by expanding quantifiers gradually but at the cost of multiple SAT calls. This approach was pioneered by the solver RAReQS and followed by QELL [THJ15] and Ijtihad [BBH⁺18b].

At the proof level, solving by expansion corresponds to the proof system $\forall\text{Exp}+\text{Res}$ [JMS13, JM15] discussed in Section 31.4.1. Sections 31.4.2 and 31.4.4 describe the algorithms AReQS and RAReQS, where AReQS accepts only 2QBF inputs and RAReQS extends AReQS by recursion to an arbitrary number of levels. The algorithms are related to the $\forall\text{Exp}+\text{Res}$ system in Sections 31.4.3 and 31.4.5.

31.4.1. The Calculus $\forall\text{Exp}+\text{Res}$

The $\forall\text{Exp}+\text{Res}$ calculus provides a transformation that enables refuting QBF by classical propositional resolution [JMS13, JM15]. This transformation stems from the idea that the existential player needs to respond to any assignment to the universal variables, but, the player may respond differently to different assignments. To represent the different choices of the existential player, $\forall\text{Exp}+\text{Res}$ annotates existential variables by the relevant assignments to the universals.

The calculus is defined in Figure 31.9. It comprises two rules. The *Axiom rule* introduces new annotated clauses into the proof and the *Res rule* enables

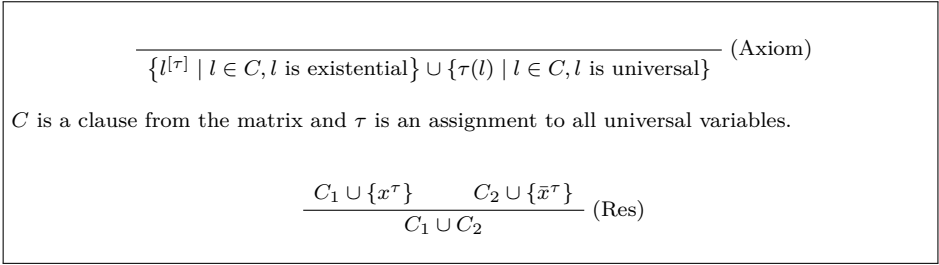


Figure 31.9. The rules of $\forall\text{Exp}+\text{Res}$ (adapted from [JM15]). The annotation $x^{[\tau]}$ considers only the universal variables that precede x in the prefix.

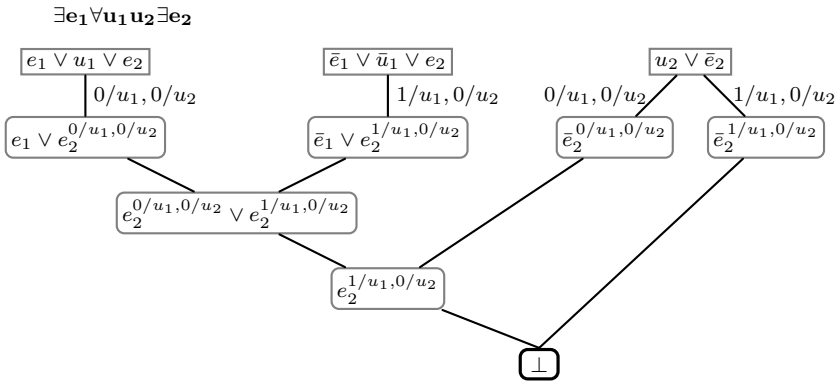


Figure 31.10. An example refutation in $\forall\text{Exp}+\text{Res}$

resolving them. An *annotated clause* is obtained by choosing a clause C from the matrix and an assignment τ to all the universal variables. The clause C is then transformed by applying τ to it and annotating each existential variable by $[\tau]$. The operation $x^{[\tau]}$ annotates x with the assignment τ restricted to the universal variables preceding x in the quantifier prefix. The rest of the proof is carried out by resolving the annotated clauses. A proof in $\forall\text{Exp}+\text{Res}$ that derives the empty clause, is called a *refutation*.

Note that the Axiom rule of the calculus potentially enables introducing an exponential number of annotated clauses. However, if the rule is applied to some assignment τ and clause C such that τ assigns some of the literals of C to true, the clause is reduced to true, denoted as \top ; such clause is useless for the rest of the proof. Observe also that since τ always assigns all universal variables, there are no universal literals in the annotated clauses. At the end of this chapter, Section 31.7 discusses more general proof systems that enable assigning only some universal variables.

Example 9. Figure 31.10 represents a possible refutation $\forall\text{Exp}+\text{Res}$ of the formula

$$\exists e_1 \forall u_1 u_2 \exists e_2 ((e_1 \vee u_1 \vee e_2) \wedge (\bar{e}_1 \vee \bar{u}_1 \vee e_2) \wedge (e_2))$$

The matrix clauses are depicted as rectangles and the annotated clauses as rectangles with rounded corners. The example demonstrates some distinctive features of the calculus. The clause $u_2 \vee \bar{e}_2$ is expanded in two different ways. The annotated clause $e_2^{0/u_1,0/u_2} \vee e_2^{1/u_1,0/u_2}$ contains two versions of the variable e_2 . Intuitively, this clause tells us that e_2 should be true in one of the two places, when both u_1 and u_2 are false or if u_1 is true and u_2 is false.

31.4.2. AReQS: Expansion-based solving for 2-level QBF

This section presents the algorithm AReQS [JMS11], which decides formulas with two levels of quantification, i.e., $\forall X \exists Y \phi$ or $\exists X \forall Y \phi$. The algorithm enables natively solving formulas whose matrix is not necessarily in CNF and we present it as such. However, in order to connect to the proof system $\forall \text{Exp} + \text{Res}$, we restrict ourselves to PCNF formulas.

To simplify the presentation we will only be looking at formulas of the form $\forall X \exists Y \phi$. The case $\exists X \forall Y$ is symmetric since $\forall X \exists Y \phi$ is true if and only if $\exists X \forall Y \neg \phi$ is false. In some scenarios, we need to call a SAT solver with non-CNF input. While SAT solvers in practice often require CNF input, this represents only a minor obstacle as CNF can be achieved by the addition of fresh variables, cf. [Tse68, PG86].

The algorithm AReQS is presented as a search for the winning move for the given formula anchored in the game perspective (see Section 31.2). Finding a winning move for a prefix of the form $\forall \exists$ shows the formula to be false, conversely, demonstrating the absence of such move shows the formula true.

Definition 6 (winning move). An assignment τ to variables X is a *winning move* for a QBF $\forall X \Phi$ if $\Phi[\tau]$ is false and it is a *winning move* for $\exists X \Phi$ if $\Phi[\tau]$ is true.

Calculating a winning move is conceptually simple when given a formula with a single quantifier block. Such formulas represent a 1-move game, where only one of the players is allowed to make a single move upon which the game terminates. Deciding such game translates naturally to propositional satisfiability. In particular, there exists a winning move for the formula $\exists X \phi$ if and only if ϕ is satisfiable. Analogously, there exists a winning move for the \forall -player for the formula $\forall X \phi$ if and only if $\neg \phi$ is satisfiable. Further, the satisfying assignments of the respective propositional formula are winning moves of the original formula.

This observation motivates the following approach to solving QBF. Start eliminating quantifiers until only one type of quantifier is left, at which point invoke a SAT solver. The question is how to eliminate quantifiers. The approach taken here is by *expansion* into Boolean connectives. This is enabled by the observation that the formula $\forall X \exists y \Phi$ has the same set of winning moves as the formula $\forall X (\Phi[0/y] \vee \Phi[1/y])$.

A key insight is that a full expansion is not necessarily needed and that a *partial expansions* may decide the formula. This is possible because assigning a variable to a specific value *reduces* the set of winning moves, cf. [JKMC16, Sec. 3].

Example 10. Let $\phi = (u \vee e_1) \wedge (\bar{u} \vee e_2) \wedge (\bar{e}_1 \vee \bar{e}_2)$ and consider $\forall u \exists e_1 e_2 \phi$. The formula is true and therefore there is no winning move for the \forall player. One

Algorithm 1: CEGAR Algorithm for 2QBF

```

input   :  $\forall X \exists Y (\phi)$ 
output :  $\tau$  if there exists a winning move  $\tau$  for  $\forall X$ ,  $\perp$  otherwise
1  $\omega \leftarrow \emptyset$ 
2 while true do
3    $\alpha \leftarrow \bigvee_{\mu \in \omega} \phi[\mu]$  // build abstraction
4    $\tau \leftarrow \text{SAT}(\neg \alpha)$  // find a candidate
5   if  $\tau = \perp$  then return  $\perp$  // no winning move, formula true
6    $\mu \leftarrow \text{SAT}(\phi[\tau])$  // find a countermove
7   if  $\mu = \perp$  then return  $\tau$  // no countermove, formula false
8    $\omega \leftarrow \omega \cup \{\mu\}$  // refine

```

could expand e_1 and e_2 with all the 4 possible assignments to e_1 and e_2 . However, considering only two is sufficient. Expanding by the assignments $\{1/e_1, 0/e_2\}$ and $\{e_1/0, e_2/1\}$ yields $\forall u(\phi[1/e_1, 0/e_2] \vee \phi[0/e_1, 1/e_2]) = \forall u(\bar{u} \vee u) = \forall u(1)$. Since $\forall u(1)$ has no winning moves, the original formula also does not have any and therefore it is true.

The idea of partial expansions is formalized in the following definition as an ω -abstraction.

Definition 7 (ω -abstraction). Let X, Y be finite sets of variables and ω be a set of assignments to Y . The ω -abstraction of a closed QBF $\forall X \exists Y \phi$ is $\forall X \bigvee_{\mu \in \omega} \phi[\mu]$.

The algorithm AREQS starts by an empty abstraction and strengthens it gradually using the *Counterexample Guided Abstraction Refinement* paradigm (CEGAR) [CGJ⁺03]. The ω -abstraction is an abstraction in the sense that its set of winning moves is a superset of the winning moves of the original formula. Adding more assignments to the set ω restricts the set of winning moves, i.e. it *refines* the abstraction.

Algorithm 1 shows the AREQS ideas in pseudocode. It consists of a loop, which in each iteration first finds a winning move for the current ω -abstraction and then it tests whether that move is also a winning move of the formula being solved. If it is, we are done. If, however, the winning move for the abstraction is *not* a winning move of the original formula, the abstraction is refined.

A winning move of the abstraction is referred to as a *candidate*. A candidate τ is a winning move of the original $\forall X \exists Y \phi$ if and only if $\exists Y \phi[\tau]$ is false. Equivalently, if $\phi[\tau]$ is unsatisfiable. Consequently, if τ is *not* a winning move, there exists an assignment μ satisfying $\phi[\tau]$; the assignment μ is called a *countermove* to the candidate move τ .

The algorithm maintains a set of countermoves ω , initialized to the empty set (line 1). In each iteration of the loop it first constructs an abstraction according to Definition 7 (line 3). Note that upon initialization, the set ω is empty and therefore $\alpha = 0$ when first entering the loop. Subsequently, the algorithm tries to find a winning move for the abstraction (line 4). If no candidate is found, it means that there is no winning move for the given formula and the algorithm terminates. If a candidate τ is found, a SAT solver is used to find a countermove to it (line 6). If there is no countermove, then τ is indeed a winning move and

the algorithm terminates. If there is a countermove μ , this countermove is added to the set ω (line 8).

Example 11. Consider the prefix $\forall u_1 u_2 u_3 \exists e_1 e_2 e_3$ and the following matrix

$$\begin{aligned} u_1 \vee u_2 \vee e_1 \vee e_2 \\ u_1 \vee u_3 \vee \bar{e}_1 \vee e_2 \\ \bar{u}_1 \vee u_2 \vee e_3 \\ \bar{u}_2 \vee u_3 \vee \bar{e}_1 \vee e_3 \end{aligned}$$

The following is a possible run of the algorithm. Let $\tau_1 = \{0/u_1, 0/u_2, 0/u_3\}$ be the first candidate with $\phi[\tau_1] = \{e_1 \vee e_2, \bar{e}_1 \vee e_2\}$. The assignment $\mu_1 = \{0/e_1, 1/e_2, 0/e_3\}$ is a countermove to τ_1 . The ω -abstraction refined by τ_1 is $\forall u_1 u_2 u_3 (\bar{u}_1 \vee u_2)$. The assignment $\tau_2 = \{1/u_1, 0/u_2, 0/u_3\}$ is a winning move for the abstraction and is chosen as the next candidate. We have $\phi[\tau_2] = \{e_3\}$, satisfiable by $\mu_2 = \{0/e_1, 0/e_2, 1/e_3\}$ chosen as the countermove.

Refining by μ_2 yields the ω -abstraction $\forall u_1 u_2 u_3 ((\bar{u}_1 \vee u_2) \vee (u_1 \vee u_2))$ which has no winning moves as its matrix is a tautology. This means that the original formula also has no winning moves and therefore it is true.

Example 11 uncovers an interesting property of AReQS in CNF. The assignments to the universal variables effectively select or deselect clauses for the existential quantifier to satisfy. The solvers QESTO [Jan18b] and CAQE [RT15] develop this idea for arbitrary number of levels. The solvers QuAbs [Ten16] and CQESTO [Jan18a] develop the ideas even further to non-CNF.

Since the winning moves of the ω -abstraction over-approximate the winning moves of the original formula, and the algorithm deems the formula true only if the abstraction has no winning moves, the original formula in such case must also be true. Termination follows from the observation that no candidates may repeat. Indeed, if μ is a countermove to τ , after the refinement the abstraction is of the form $\forall X(\alpha' \vee \phi[\mu])$. Consequently, any subsequent candidate τ' must falsify $\phi[\mu]$. This means that τ' cannot be τ as τ together with μ satisfy ϕ . By a similar argument we can show that countermoves also do not repeat.

31.4.3. Connecting AReQS to $\forall\text{Exp}+\text{Res}$

In order to relate AReQS to $\forall\text{Exp}+\text{Res}$, we focus only on the refutation of formulas of the form $\forall X \exists Y \phi$ where ϕ is in CNF. If $\forall X \exists Y \phi$ is false, there must be a winning move for the universal player. This means that AReQS finds an assignment τ to the first block such that there is no countermove to it. More specifically, $\phi[\tau]$ is unsatisfiable. Since $\phi[\tau]$ is unsatisfiable in purely propositional sense, there also exists a propositional resolution refutation π for it. The $\forall\text{Exp}+\text{Res}$ refutation for the original formula $\forall X \exists Y \phi$ is constructed by following the structure of π .

Each leaf of the resolution refutation π is a clause $C' \in \phi[\tau]$ obtained from some clause $C \in \phi$. Construct a clause C^τ by invoking the Axiom rule on C using the assignment τ . The Axiom rule removes all universal variables from C and annotates each existential variable with τ . The rest of the proof is constructed by applying the same resolution steps as in π : whenever there is a resolution step on x in π , perform a resolution step on x^τ .

Applying the Axiom rule parameterized by the substitution τ to the clauses of the matrix, yields annotated clauses corresponding to the clauses in $\phi[\tau]$, where each existential variable is annotated by τ .

Example 12. Consider the clauses $C_1 = u_1 \vee e_1$, $C_2 = u_2 \vee \bar{e}_1$, and $C_3 = \bar{u}_1 \vee e_2$ under the prefix $\forall u_1 u_2 \exists e_1 e_2$. The matrix becomes unsatisfiable under the assignment $\tau = \{0/u_1, 0/u_2\}$. Substituting τ into the matrix gives $C'_1 = e_1$, $C'_2 = \bar{e}_1$, and $C'_3 = \top$, where unsatisfiability is obtained by resolving C'_1 and C'_2 . The corresponding $\forall\text{Exp}+\text{Res}$ proof applies the Axiom rule on C_1 and C_2 yielding $C_1^\tau = e_1^\tau$, $C_2^\tau = \bar{e}_1^\tau$, which are then resolved into \perp .

31.4.4. RReQS: Expansion-based QBF Solving

This section presents the algorithm RReQS [JKMSC12], which generalizes AReQS to an arbitrary number of quantification levels by recursion. The algorithm tries to find a winning move for the given formula but rather than calling a SAT solver, it uses recursive calls as a subroutine. The base case, with one quantifier block left, is handled by a SAT solver.

Analogously to AReQS quantifiers are expanded into a propositional operator (disjunction of conjunction) and a partial expansion lets us approximate a set of winning moves. An abstraction of a QBF with at least two quantifier blocks is obtained by partially expanding the second quantifier block. Just as before, a *candidate* refers to a winning move of an abstraction.

Applying expansion to arbitrary quantifiers requires additional care. Consider for instance $\exists x \forall y \exists z \phi$. Expanding the middle quantifier $\forall y$ into a conjunction yields a QBF in non-prenex form: $\exists x (\exists z \phi[0/y] \wedge \exists z \phi[1/y])$. Therefore, the abstraction is calculated by additionally prenexing the expanded formula. We write $\text{prenex}(\Phi)$ for the prenexed form of a QBF Φ . Prenexing may require fresh variables. For instance, $\text{prenex}(\exists x (\exists z \phi[0/y] \wedge \exists z \phi[1/y]))$ gives $\exists x z^0 z^1 (\phi[0/y, z^0/z] \wedge \phi[1/y, z^1/z])$. We will see later that these fresh variables correspond to the annotations in $\forall\text{Exp}+\text{Res}$.

Definition 8 (ω -abstraction). Let X, Y be finite sets of variables and ω be a set of assignments to Y .

The ω -abstraction of a closed QBF $\forall X \exists Y \Phi$ is the formula $\text{prenex}(\forall X \bigvee_{\mu \in \omega} \Phi[\mu])$.

The ω -abstraction of a closed QBF $\exists X \forall Y \Phi$ is the formula $\text{prenex}(\exists X \bigwedge_{\mu \in \omega} \Phi[\mu])$.

The algorithm is presented as a recursive function `solve`. Its implementation for the existential quantifier is presented in Algorithm 2. The implementation for the universal quantifier is analogous. The function accepts a closed QBF $QX\Phi$ where $Q \in \{\forall, \exists\}$; it returns a winning move for QX , if such exists and it returns \perp otherwise.

The base case of the recursion is when Φ does not contain any quantifiers, in which case a SAT solver is used to find a winning move. In the general case, an ω -abstraction is gradually strengthened by countermoves in a similar vein as in AReQS. The prenexing step causes a small technical difficulty. A winning move for the abstraction potentially contains values for variables that correspond to copies of variables originally appearing in inner quantifiers. These need to be filtered out as they do not exist in the original formula (line 10).

Algorithm 2: Basic recursive CEGAR for QBF (existential player)

```

1 Function solve( $\exists X\Phi$ )
  input   :  $\exists X\Phi$  is a closed QBF in prenex form with no adjacent blocks with the
             same quantifier
  output  : a winning move for  $\exists X\Phi$  if there is one,  $\perp$  otherwise
2 begin
3   if  $\Phi$  has no quantifiers then
4     return SAT( $\Phi$ )
5    $\omega \leftarrow \emptyset$ 
6   while true do
7      $\alpha \leftarrow \bigwedge_{\mu \in \omega} \Phi[\mu]$ 
8      $\tau' \leftarrow \text{solve}(\text{prenex}(\exists X\alpha))$  // find a candidate solution
9     if  $\tau' = \perp$  then return  $\perp$  // no winning move, formula true
10     $\tau \leftarrow \{c/v \mid \tau'(x) = c \text{ and } v \in X\}$  // filter a move for  $X$ 
11     $\mu \leftarrow \text{solve}(\Phi[\tau])$  // find a countermove
12    if  $\mu = \perp$  then return  $\tau$  // found winning move, formula false
13     $\omega \leftarrow \omega \cup \{\mu\}$  // refine

```

Observe that in each iteration there are two recursive calls. First call obtains a winning move for the abstraction—a candidate (line 8). The second call checks whether the candidate is a winning move or not (line 11). In the first call the number of quantifier levels drops by two because the second quantifier is eliminated by expansion and the third quantifier is merged with the first one during prenexing. In the second call, the number of quantifier levels drops by one.

Example 13. Consider the QBF $\exists v w \Phi$, where

$$\Phi = \forall u \exists x y ((v \vee w \vee x) \wedge (\bar{v} \vee y) \wedge (\bar{w} \vee y) \wedge (u \vee \bar{x}) \wedge (\bar{u} \vee \bar{y})),$$

The candidates $\{1/v, 1/w\}$ and $\{0/v, 0/w\}$ give the counterexamples $\{1/u\}$ and $\{0/u\}$, respectively. Refinement yields the expansion $\exists v w (\Phi[0/u] \wedge \Phi[1/u])$, with the prenex form

$$\begin{aligned} \exists v w x^0 y^0 x^1 y^1 ((v \vee w \vee x^0) \wedge (\bar{v} \vee y^0) \wedge (\bar{w} \vee y^0) \wedge (\bar{x}^0) \\ \wedge (v \vee w \vee x^1) \wedge (\bar{v} \vee y^1) \wedge (\bar{w} \vee y^1) \wedge (\bar{y}^1)) \end{aligned}$$

At this point, the abstraction is purely propositional and it is unsatisfiable. Hence, it has no winning move and the function terminates with the return value \perp , which means that the formula is false.

The arguments for correctness and termination are analogous to AReQS. Candidate moves or counterexamples cannot repeat. If in any of the iterations of the loop, a candidate τ was countered by some μ , the move τ will never be drawn from the abstraction because the refinement guarantees that τ is losing in the strengthened abstraction.

RAREQS, compared to the presentation here implements a modification that mitigates the growth of the abstraction for larger number of qualification levels (starting at 4). Rather than prenexing the whole prefix, only the outermost quantifier is prenexed and the rest is maintained in non-prenex form. For instance,

$\forall X \exists Y \forall Z \exists W \phi$ may be expanded as $\forall X Z^1 Z^2 ((\exists W \phi^1) \vee (\exists W \phi^2))$. Since \forall distributes over disjunction, any winning move for \forall must be a winning move for each of the individual disjuncts. Like so, candidate tests and refinements are carried out for each disjunct separately. From the game-theoretical point of view, each disjunct can be seen as a separate game that the quantifier must win. Therefore the approach is referred to as *multi-games* [JKMC16, Sec 4.1]. The existential case is again analogous.

31.4.5. Connecting RAReQS to $\forall\text{Exp}+\text{Res}$

We show how to produce $\forall\text{Exp}+\text{Res}$ refutations (Figure 31.9) from Algorithm 2. Consider the following naming scheme for fresh variables used in prenexing. As a part of the name of each existential variable is a annotation corresponding to a substitution to some universal variables. In the input formula these annotations are empty. The annotation is extended whenever a fresh copy of the variable is added to an abstraction. Like so, as the recursion proceeds, the annotations are getting longer and at the same time the universal variables disappear from the formula.

Consider the prefix $\exists E_1 \forall U_2 \exists E_3 \dots \exists E_n \phi$. If we are given a formula that starts by the universal quantifier, add a dummy $\exists E_1$. The expansion by some assignments μ_1 and μ_2 to the variables U_2 corresponds to the following transformations.

$$\exists E_1 ((\exists E_3 \dots \exists E_n \phi[\mu_1]) \wedge (\exists E_3 \dots \exists E_n \phi[\mu_2])) \quad (31.1)$$

$$\exists E_1 E_3^{\mu_1} E_3^{\mu_2} \dots \exists E_n^{\mu_1} \exists E_n^{\mu_2} (\phi[\mu_1, E_3^{\mu_1}/E, \dots, E_n^{\mu_1}/E_n] \wedge \phi[\mu_2, E_3^{\mu_2}/E, \dots, E_n^{\mu_2}/E_n]) \quad (31.2)$$

The original formula is expanded in (31.1) and prenexed in (31.2). In (31.2), X^μ denotes a set of variables obtained by appending μ to the annotation of each variable in X . Since ϕ is in CNF, the matrix of (31.2) is also in CNF. At the same time the clauses observe the invariant that each clause that comes from the expansion by μ_i has μ_i in its annotations.

The base case of the recursion of Algorithm 2 takes place when there are no universal variables left in the matrix. This means that in such case the matrix contains annotated clauses containing only existential variables, as required by the expansion rule in the calculus. These are handed to a SAT solver. If the expansion is false, the SAT solver deems the expansion unsatisfiable, which is then certified by propositional resolution refutation. This refutation is readily translated to a $\forall\text{Exp}+\text{Res}$ refutation as each propositional resolution step corresponds to a $\forall\text{Exp}+\text{Res}$ resolution step.

31.5. Preprocessing

Preprocessing [BLS11, BK07, GMN10, LE18, SDB06, VGWL12, WRMB17] a QBF before it is passed to a complete solver is beneficial in many cases [LSVG16], Preprocessing is particularly helpful when the solver does not try do reconstruct information that was lost by the transformation to prenex conjunctive normal

form (cf. [JKMC16, GB13] for such solvers). Preprocessors like sQueezBF [GMN10], Bloqqer [BLS11], HQSpre [WRMB17], or QRATPre+ [LE18] apply rewriting techniques that preserve (satisfiability) equivalence and that modify formulas in such a way such that information relevant for the solver becomes easier to access and such that irrelevant information is eliminated.

31.5.1. Background

There are three types of preprocessing rules: (1) clause elimination rules, (2) clause addition rules, and (3) clause modification rules. The application of these rules might also require a modification of the prefix, because new variables could be introduced while some variables might vanish.

Clause elimination rules [HJL⁺15] remove clauses while preserving unsatisfiability. Examples of such rules are tautology elimination, subsumption, the existential pure literal rule, and blocked clause elimination. Tautology elimination removes clauses containing a positive and negative occurrence of a variable. Subsumption [Bie04] removes clauses that are a superset of other clauses. The existential pure literal rule [CGS98a] removes all clauses with an existential literal that occurs only positively or only negatively in the formula (there is also a universal pure literal rule, but as being a clause modification rule, it is discussed below). Blocked clause elimination [BLS11] removes a clause C from a QBF $\Pi\psi$ if C contains an existential literal l such that for all $D \in \psi$ with $\bar{l} \in D$, there exists a literal k with $k \leq_{\Pi} l$ such that $k, \bar{k} \in C \cup D \setminus \{\bar{l}\}$, i.e., the clause $C \cup D \setminus \{\bar{l}\}$ is a tautology.

Example 14. Consider the QBF ϕ given by

$$\forall x_1 x_2 \exists y_1 y_2 ((x_1 \vee y_1) \wedge (y_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{y}_1) \wedge (\bar{y}_2 \vee y_1) \wedge (x_1 \vee y_1 \vee x_2) \wedge (\bar{x}_1 \vee x_1 \vee y_2)).$$

The last clause is a tautology, hence it can be safely removed. After the removal of the last clause, the existential variable y_2 occurs only in one polarity, therefore the existential pure literal elimination rule is applicable and clause $(\bar{y}_2 \vee y_1)$ can be removed. Also x_2 occurs only in one polarity, but since it is universally quantified, we are not allowed to remove clause $(x_1 \vee y_1 \vee x_2)$ with the existential pure literal rule. Nevertheless, we can remove this clause, because it is subsumed by $(x_1 \vee y_1)$. Next, we eliminate clause $(x_1 \vee y_1)$, because it is blocked on y_1 . Finally, we obtain the simplified QBF $\forall x_1 x_2 \exists y_1 ((y_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{y}_1))$ which is equivalent to ϕ .

Clause addition rules enrich a formula with new clauses besides modifying and removing clauses. Variable elimination [Bie04] replaces the clauses in which a certain existential variable occurs by all non-tautological resolvents on that variable. Therefore, the *res*-Rule as introduced in Section 31.3.1 is applied on the existential variables of the innermost quantifier block given that a certain limit of additional clauses is not reached.

Partial universal expansion [Bie04, BK07] is a restricted form of universal expansion as discussed in Section 31.4. It eliminates a universal variable x of the innermost universal quantifier block by copying the matrix of the QBF. Then x is set to true in one copy and to false in the other copy. For combing these two CNFs, in one copy new names must be assigned to the variables of the innermost existential quantifier block.

Example 15. Consider the QBF $\forall x_1 x_2 \exists y_1 ((y_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{y}_1))$ from the previous example. Eliminating y_1 results in QBF $\forall x_1 x_2 ((x_2 \vee \bar{x}_1))$. The universal expansion of the universal variables derives a formula containing the empty clause. If we expand x_1 in $\forall x_1 x_2 \exists y_1 ((y_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{y}_1))$, we obtain $\forall x_2 \exists y_1 y_1^0 ((y_1 \vee x_2) \wedge (\bar{y}_1) \wedge (y_1^0 \vee x_2))$. Note that y_1 was renamed to y_1^0 in the case that x_1 was set to false, for the case that x_1 was set to true, we keep y_1 . Eliminating variable x_2 finally results in $\exists y_1 y_1^{00} y_1^{01} ((\bar{y}_1) \wedge (y_1^{00}) \wedge (\bar{y}_1^{00}) \wedge (y_1^{01}))$.

Clause modification rules add, remove, and rename literals. One important rule is universal reduction, the *red-Rule*, as introduced above in Section 31.3.1. Strengthening [Bie04] removes a literal l from a clause C if there is another clause $D \vee \bar{l}$ in the formula such that D subsumes C . The unit literal elimination rule [CGS98a] is applicable if a QBF $\Pi\psi$ contains a clause C with exactly one existential literal l and for all universal literals $k \in C$ $l \leq_{\Pi} k$. Then all clauses containing l are removed and so are all occurrences of \bar{l} . Universal pure literal [CGS98a] elimination removes all occurrences of a universal literal l if it occurs only in one polarity in the considered QBF. Covered literal addition adds literals to a clause C that occur in all non-tautological resolvents obtained from C . The equivalence replacement rule [Bie04] can be applied in QBF $\Pi\psi$ when two literals l, k with l is existentially quantified and $k \leq_{\Pi} l$ are found to be equivalent by detecting the clauses $(l \vee k)$ and $(\bar{l} \vee k)$ in ψ . Finally, the blocked literal [HSB15] elimination rule can remove a universal literal l from a clause C if all resolvents on l contain two opposite literals k and \bar{k} with $k \leq_{\Pi} l$. Blocked literal elimination generalizes the universal pure literal rule.

Example 16. Consider the QBF

$$\forall x_1 x_2 \exists y_1 y_2 ((y_1 \vee x_2 \vee \bar{y}_2) \wedge (x_1 \vee \bar{y}_2) \wedge (\bar{x}_1 \vee y_2) \wedge (\bar{y}_1 \vee \bar{x}_1 \vee \bar{y}_2)).$$

By strengthening, the clause $D = (\bar{y}_1 \vee \bar{x}_1 \vee \bar{y}_2)$ can substituted by $(\bar{y}_1 \vee \bar{x}_1)$, because its resolvent x_1 which obtained by resolving D on \bar{y}_2 with clause $(\bar{x}_1 \vee y_2)$ subsumes D . By equivalence replacement, we may substitute y_2 by x_1 and under omission of the two tautological clauses, we get $\forall x_1 x_2 \exists y_1 ((y_1 \vee x_2 \vee \bar{x}_1) \wedge (\bar{y}_1 \vee \bar{x}_1))$. Now x_1 and x_2 are pure literals and can therefore be omitted. The application of unit literal elimination results in a conflict.

For almost all of these preprocessing rules, Q-resolution proofs (see Section 31.3) can be generated during their application [JGM13] to witness unsatisfiability. However, this approach does not work for universal expansion which is a very crucial technique in preprocessing. In order to capture all recent preprocessing techniques including universal expansion, the QRAT proof system was developed.

31.5.2. The QRAT Proof System

The QRAT proof system [HSB17] lifts the RAT proof system (see Chapter 15) from SAT to QBF. The RAT proof system introduces rules for adding and removing clauses such that the satisfiability of a formula is preserved. Besides the QBF variants of clause addition and clause deletion rules that have to take the ordering

imposed by the prefix into account, the QRAT proof system provides additional rules for clause modification. Before we introduce the rules of QRAT, we first define the notions of *outer resolvent* and *implication via unit propagation*.

Definition 31.5.1. The *outer resolvent* of clauses $C \vee l$ and $D \vee \bar{l}$ on literal l w.r.t. prefix Π is defined as $\text{OR}(\Pi, C, D, l) = C \cup \{k \mid k \in D, k \leq_{\Pi} l, k \neq l\}$.

In contrast to the classical resolvent $C \cup D$, the outer resolvent contains only those literals of D that occur earlier in the prefix than the pivot literal l .

Example 17. Given prefix $\Pi = \forall x_1 \exists y_1 \forall x_2 \exists y_2 y_3$ and clauses $C = (y_1 \vee \bar{x}_2 \vee y_3)$ and $D = (x_1 \vee \bar{y}_1 \vee y_3)$ the outer resolvent $\text{OR}(\Pi, C, D, y_1) = (y_1 \vee \bar{x}_2 \vee y_3 \vee x_1)$. The outer resolvent $\text{OR}(\Pi, D, C, \bar{y}_1) = (x_1 \vee \bar{y}_1 \vee y_3)$.

Proof systems like resolution and Q-resolution derive clauses that—when added to the formula under consideration—preserve logical equivalence. The QRAT proof system is an interference-based proof system, i.e., the addition of newly derived clauses or the elimination of a redundant clause only preserves satisfiability equivalence meaning that the set of models/counter-models might change. As in RAT, the concept of reverse unit propagation is used to justify clause addition or deletion.

Definition 31.5.2. A propositional formula ψ *implies a clause* $(l_1 \vee \dots \vee l_n)$ *via unit propagation* (denoted by $\psi \vdash_1 C$) iff unit propagation derives the empty clause \perp on $\psi \wedge \bar{l}_1 \wedge \dots \wedge \bar{l}_n$ where unit propagation refers to the application of the unit literal elimination rule until fixpoint.

A clause that is implied by unit propagation is sometimes also called an *asymmetric tautology* [HJL⁺15] or a clause that has the redundancy property of reverse unit propagation (RUP) [VG12b].

Example 18. Consider the propositional formula $(x \vee y) \wedge (\bar{x} \vee y)$. The clause $(x \vee y)$ is an asymmetric tautology, because for $((x \vee y) \wedge (\bar{x} \vee y) \wedge \bar{x} \wedge \bar{y})$ unit propagation derives a conflict.

Based on outer resolvents, we introduce the *quantified asymmetric tautology* property (QRAT) next.

Definition 31.5.3. A clause C has QRAT on a literal $l \in C$ w.r.t. QBF $\Pi\psi$ iff $\psi \vdash_1 \text{OR}(\Pi, C, D, l)$ for all $D \in \psi$ with $\bar{l} \in D$.

If a clause C has QRAT on literal l w.r.t. $\Pi\psi$, we call l a QRAT-literal of C for $\Pi\psi$. Note that a clause C that is implied via unit propagation by a propositional formula ψ has QRAT because all outer resolvents built from C contain C and are therefore implied via unit propagation as well. The following properties of QRAT-literals proven in [HSB17] finally allow us to formulate the rules of the QRAT proof system (see Figure 31.11). If a clause contains an existential QRAT-literal for a QBF $\Pi\psi$ then clause C is redundant in $\Pi\psi$, i.e., $\Pi\psi$ and $\Pi(\psi \cup \{C\})$ are satisfiability equivalent. If a clause C contains a universal QRAT-literal for QBF $\Pi\psi$, then l is redundant in C , i.e., $\Pi(\psi \cup \{C\})$ and $\Pi(\psi \cup \{C \setminus \{l\}\})$ are satisfiability equivalent. Note that the clause and literal addition rules can introduce new variables that have to be included in the prefix.

Rule	Rule Application	Side Conditions
Clause Elimination	$\Pi(\psi \cup \{C\}) \xrightarrow{\text{QRATE}(C, l)} \Pi\psi$	<ul style="list-style-type: none"> • l is existential or $l = \perp^*$, $l \in C$ • l is a QRAT-literal of C for $\Pi\psi$
Clause Addition	$\Pi\psi \xrightarrow{\text{QRATA}(C, l)} \Pi'(\psi \cup \{C\})$	<ul style="list-style-type: none"> • l is existential or $l = \perp^*$, $l \in C$ • l is a QRAT-literal of C for $\Pi'(\psi)$ • Π' is obtained from Π by adding the variables only occurring in C <p>*if $l = \perp$ then C is an asymmetric tautology</p>
Literal Elimination	$\Pi(\psi \cup \{C\}) \xrightarrow{\text{QRATE}(C, l)} \Pi(\psi \cup \{C'\})$	<ul style="list-style-type: none"> • l is universal, $l \in C$, $C' = C \setminus \{l\}$ • l is a QRAT-literal of C for $\Pi\psi$
Literal Addition	$\Pi(\psi \cup \{C\}) \xrightarrow{\text{QRATA}(C, l)} \Pi'(\psi \cup \{C'\})$	<ul style="list-style-type: none"> • l is universal, $C' = C \cup \{l\}$, $l \notin C$ • l is a QRAT-literal of C for $\Pi'\psi$ • Π' is obtained from Π plus the variable of l if it is new
Extended Universal Reduction	$\Pi(\psi \cup \{C\}) \xrightarrow{\text{EUR}(C, l)} \Pi(\psi \cup \{C'\})$	<ul style="list-style-type: none"> • l is universal, $l \in C$, $C' = C \setminus \{l\}$ • for all $k \in C$ with $k \supset_{\Pi} l$, l and k are independent w.r.t. the resolution-path dependency scheme

Figure 31.11. Rules of the QRAT Proof System

Besides the clause addition, clause elimination, literal addition, and literal elimination rules based on the QRAT property, there is a generalized version of universal reduction included that is based on resolution-path dependency schemes (see also Section 31.3.2). Given a QBF ϕ_0 , a derivation in the QRAT proof system is a sequence M_1, \dots, M_n where M_i is one of the rules shown in Figure 31.11 applied on QBF ϕ_{i-1} to obtain QBF ϕ_i . Hence, starting from a given QBF $\phi = \phi_0$, a QRAT derivation M_1, \dots, M_n produces new, equi-satisfiable formulas ϕ_1, \dots, ϕ_n . If ϕ_n contains the empty clause, then the derivation is a *proof of unsatisfiability*. If the matrix of ϕ_n is empty, then the derivation is a *proof of satisfiability*.

Example 19. Consider the following true QBF $\Pi\psi$ with $\Pi = \forall x \exists y_1 y_2$ with $\psi = ((x \vee y_1) \wedge (\bar{x} \vee y_2) \wedge (\bar{y}_1 \vee \bar{y}_2))$. It has the following QRAT proof of satisfiability:

$\Pi\psi$	$\frac{\text{QRATA}((\bar{y}_1 \wedge \bar{x}), \bar{y}_1)}{\text{QRATE}((y_2 \wedge \bar{x}), y_2)}$	$\Pi(\psi \cup \{(\bar{y}_1 \wedge \bar{x})\})$ $\Pi(((x \vee y_1) \wedge (\bar{y}_1 \vee \bar{y}_2) \wedge (\bar{y}_1 \wedge \bar{x}))$	$(\bar{y}_1 \wedge \bar{x})$ is blocked on \bar{y}_1 OR ($\Pi, (y_2 \wedge \bar{x}), (\bar{y}_1 \vee \bar{y}_2), y_2$) has RUP
	$\frac{\text{QRATE}((\bar{y}_2 \wedge \bar{y}_1), \bar{y}_2)}{\text{QRATE}((\bar{y}_1 \wedge \bar{x}), \bar{y}_1)}$	$\Pi(((x \vee y_1) \wedge (\bar{y}_1 \wedge \bar{x}))$ $\Pi((x \vee y_1))$	$(\bar{y}_2 \wedge \bar{y}_1)$ is blocked on \bar{y}_2 $(\bar{y}_1 \wedge \bar{x})$ is blocked on \bar{y}_1
	$\frac{\text{QRATE}((y_1 \wedge x), y_1)}{\text{QRATE}((y_1 \wedge x), y_1)}$	$\Pi(\top)$	$(y_1 \wedge x)$ is blocked on y_1

Example 20. Consider the following false QBF $\Pi\psi$ with $\Pi = \forall x \exists y_1 y_2$ with $\psi = ((x \vee y_1) \wedge (x \vee y_2) \wedge (\bar{y}_1 \vee \bar{y}_2))$. It has the following QRAT proof:

$\Pi\psi$	$\frac{\text{QRATA}((\bar{y}_1), \bar{y}_1)}{\text{QRATE}((\bar{y}_1 \vee \bar{y}_2), \perp)}$	$\Pi(\psi \cup \{(\bar{y}_1)\})$ $\Pi((x \vee y_1) \wedge (x \vee y_2) \vee (\bar{y}_1))$	OR ($\Pi, \bar{y}_1, (x \vee y_1), \bar{y}_1$) = $(x \vee \bar{y}_1)$ has RUP $(\bar{y}_1 \vee \bar{y}_2)$ has RUP
	$\frac{\text{QRATE}((x), \perp)}{\text{EUR}((x), x)}$	$\Pi((x \vee y_1) \wedge (x \vee y_2) \vee (\bar{y}_1) \wedge (x))$ $\Pi((x \vee y_1) \wedge (x \vee y_2) \vee (\bar{y}_1) \wedge (\perp))$	(x) has RUP universal reduction de- rives \perp

31.5.3. QRAT and Its Extension QRAT⁺ for Preprocessing

Heule et al. [HSB17] showed how to express preprocessing techniques in terms of QRAT rules, implemented QRAT proof generation in the preprocessor **Bloqper** and provided a proof checking tool. In many cases, the proof generation is straight forward. For example, a clause C that contains a pure literal l , obviously has QRAT on l . Hence, if l is existential, QRAT witnesses that C may be safely removed. If l is universal then QRAT witnesses that l may be safely removed from C . The argument generalizes to blocked clauses and blocked literals. Also tautologies and subsumed clauses have the QRAT property. For other rules the translation is more involved, because multiple different QRAT rule applications are necessary. In the following, we show an example of a QRAT proof for the restricted form universal expansion as used in preprocessing, i.e., the expanded universal variable occurs in the innermost universal quantifier block. For the details we refer to [HSB17].

Example 21. In Example 15 we have seen that for QBF $\Pi.\psi$ with $\Pi = \forall x_1 x_2 \exists y_1$ and $\psi = ((y_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{y}_1))$ universally expanding x_1 results in the formula $\forall x_2 \exists y_1 y_1^0 ((y_1 \vee x_2) \wedge (\bar{y}_1) \wedge (y_1^0 \vee x_2))$. In the following, we certify this rewriting of $\Pi.\psi$ with the following QRAT proof.

$$\begin{array}{ll}
\Pi\psi \xrightarrow{\text{QRATA}((y_1 \vee x_2 \vee \bar{x}_1), \perp)} & \Pi((y_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{y}_1) \wedge (y_1 \vee x_2 \vee \bar{x}_1)) \quad (1) \\
\xrightarrow{\text{QRATA}((\bar{y}_1^0 \vee x_1 \vee y_1), \bar{y}_1^0)} & \Pi'((y_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{y}_1) \wedge (y_1 \vee x_2 \vee \bar{x}_1) \wedge (\bar{y}_1^0 \vee x_1 \vee y_1)) \quad (2) \\
\xrightarrow{\text{QRATA}((y_1^0 \vee x_1 \vee \bar{y}_1), y_1^0)} & \Pi'((y_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{y}_1) \wedge (y_1 \vee x_2 \vee \bar{x}_1) \wedge (\bar{y}_1^0 \vee x_1 \vee y_1) \wedge (y_1^0 \vee x_1 \vee \bar{y}_1)) \quad (3) \\
\xrightarrow{\text{QRATA}((y_1^0 \vee x_2 \vee x_1), y_1^0)} & \Pi'((y_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{y}_1) \wedge (y_1 \vee x_2 \vee \bar{x}_1) \wedge (\bar{y}_1^0 \vee x_1 \vee y_1) \wedge (y_1^0 \vee x_1 \vee \bar{y}_1) \wedge (y_1^0 \vee x_2 \vee x_1)) \quad (4) \\
\xrightarrow{\text{QRATE}((y_1 \vee x_2), \perp)} & \Pi'((\bar{x}_1 \vee \bar{y}_1) \wedge (y_1 \vee x_2 \vee \bar{x}_1) \wedge (\bar{y}_1^0 \vee x_1 \vee y_1) \wedge (y_1^0 \vee x_1 \vee \bar{y}_1) \wedge (y_1^0 \vee x_2 \vee x_1)) \quad (5) \\
\xrightarrow{\text{QRATE}((\bar{y}_1^0 \vee x_1 \vee y_1), y_1)} & \Pi'((\bar{x}_1 \vee \bar{y}_1) \wedge (y_1 \vee x_2 \vee \bar{x}_1) \wedge (y_1^0 \vee x_1 \vee \bar{y}_1) \wedge (y_1^0 \vee x_2 \vee x_1)) \quad (6) \\
\xrightarrow{\text{QRATE}((y_1^0 \vee x_1 \vee \bar{y}_1), \bar{y}_1)} & \Pi'((\bar{x}_1 \vee \bar{y}_1) \wedge (y_1 \vee x_2 \vee \bar{x}_1) \wedge (y_1^0 \vee x_2 \vee x_1)) \quad (7) \\
\xrightarrow{\text{EUR}^*} & \forall x_2 \exists y_1 y_1^0 ((\bar{y}_1) \wedge (y_1 \vee x_2) \wedge (y_1^0 \vee x_2)) \quad (8)
\end{array}$$

In Step (1), the subsumed clause $(y_1 \vee x_2 \vee \bar{x}_1)$ is introduced. This is possible, because it has RUP. In Step (2) the clause $(\bar{y}_1^0 \vee x_1 \vee y_1)$ with the fresh variable y_1^0 variable is introduced. Obviously, \bar{y}_1^0 is a QRAT literal for prefix $\Pi' = \Pi \exists y_1^0$. The clause introduced in Step (3) is needed for renaming occurrences of y_1 to y_1^0 as done in Step (5). In fact, the clauses introduced in Step (3) and Step (4) form a conditional equivalence between y_1 and y_1^0 . In Step (5) and Step (6) the clauses of the conditional equivalence are removed again, because no further renaming is necessary. The removal is possible, because y_1 and \bar{y}_1 are RAT literals. Finally, all occurrences of x_1 be removed by the application of EUR and we obtain indeed the formula $\forall x_2 \exists y_1 y_1^0 ((y_1 \vee x_2) \wedge (\bar{y}_1) \wedge (y_1^0 \vee x_2))$.

Originally, QRAT was developed to provide a uniform way to certify the result of a preprocessor. As the QRAT proof system provides sound rules to add, remove and modify clauses, the QRAT rules themselves can be used within a preprocessor to modify a formula. For this purpose, even a generalization of QRAT called QRAT⁺ was developed in [LE18]. The version of QRAT as introduced above uses propositional unit propagation for testing the QRAT property (see Def 31.5.3). Here the quantification of the variables is not taken into account. QRAT⁺ uses a QBF-specific variant of unit propagation which also includes universal reduction. So certain clauses can be shortened during unit propagation resulting in a more powerful redundancy property.

31.6. Extraction of Winning Strategies from Proofs

Only few solvers explicitly generate winning strategies while evaluating a formula. For example the incremental determinization approach [RTRS18] generalizes the ideas behind CDCL (see Chapter 4 from searching for satisfying assignments to searching for Skolem functions. The solver QESTO [Jan18b] uses strategies instead of assignments for generating expanded formulas (see also Section 31.4). Most solvers, however, generate winning strategies only implicitly. For the solving approaches discussed before, proofs serve not only as witnesses for satisfiability/unsatisfiability, but they provide the necessary information for extracting a winning strategy, i.e., a model for a true QBF or countermodel for a false QBF. We distinguish between static function-extraction approaches (offline approaches)

and dynamic round-based approaches (online approaches). Both are shortly discussed in the following.

31.6.1. Function-Extraction Approaches

Balabanov and Jiang [BJ11, BJ12] presented an approach that allows for the extraction of Herbrand functions from clause Q-resolution proofs, and dually for the extraction of Skolem functions from cube Q-resolution proofs. Given a proof, the computation of such a function is linear in the size of the proof, allowing to use the simple representation format of decision lists.

Definition 9. A decision list $D = (t_1, v_1), \dots, (t_n, v_n), (\top, 0)$ is a finite sequence of pairs where t_i is a conjunction of literals and $v_i \in \{0, 1\}$. Let σ be an assignment over the variables occurring in a decision list D . Then D evaluates to v_i under σ if i is the least index such that t_i is true under σ .

Now let ϕ be a false QBF refuted by a Q-resolution proof π as described above. As ϕ is false, there is a winning strategy for the universal player. By traversing the proof in topological order, for each universal variable u , a decision list D_u is constructed that encodes such a winning strategy. Let C_i be a clause occurring in π that has been derived by removing the universal literal l from the previously derived clause C_k by the application of the universal reduction rule. If $l = u$, then the pair $(\bar{C}_i, 0)$ is appended to D_u . If $l = \bar{u}$, then the pair $(\bar{C}_i, 1)$ is appended to D_u . After the full proof has been traversed, all decision lists are closed with a pair $(\top, 0)$.

Example 22. Consider the false QBF $\exists x \forall a \exists z ((x \vee a \vee y) \wedge (\bar{x} \vee \bar{a} \vee y) \wedge \bar{y})$. A possible resolution proof first derives $(x \vee a)$ by resolving the first and the last clause. Applying universal reduction results in the clause x . Further, $(\bar{x} \wedge \bar{a})$ is then derived by resolving the second and the third clause. Applying universal reduction results in the clause \bar{x} . Another application of the resolution rule on x and \bar{x} derives the empty clause.

The proof contains two applications of the universal reduction rule from which the decision list $(\bar{x}, 0), (x, 1), (\top, 0)$ can be extracted. This decision list represents the Herbrand function $f_y(x) = x$ which is indeed a counter-model of the given formula.

This form of function extraction is also possible for long-distance Q-resolution proofs [BJJW15]. Also from QRAT proofs of satisfiability Skolem functions can be extracted by traversing the proof for constructing conditional expressions [HSB14, HSB17]. In contrast to decision lists, the conditions are not only conjunctions of literals, but more complicated formulas involving all outer resolvents of the QRAT literal of each clause that is deleted. This approach allows also the extraction of incomplete Skolem functions in case the preprocessor cannot solve a given formula. If a complete solver can solve the preprocessed formula, the Skolem functions of the preprocessed formula can be combined with the incomplete Skolem functions for obtaining the Skolem functions of the original formula [FHSB17]. For QRAT proofs of unsatisfiability, it is not known how to extract Herbrand functions so far.

31.6.2. Round-Based Approaches

An alternative approach directly implements the game semantics of QBFs. The quantifier prefix is processed from left to right. First, an arbitrary assignment is chosen by the existential player for the outermost existential variables. Based on this assignment, the proof is rewritten to a smaller refutation proof which also provides the information how the universal player has to assign the variables of the next universal quantifier block in order to win the game. Then it is again the existential player's turn to assign the variables of the next existential quantifier block. This assignment is again used to rewrite the proof, yielding the values of the next universal variables and so on. This approach works very similar for Q-resolution proofs and $\forall\text{Exp}+\text{Res}$ proofs. The main difference is in finding the values for the universal variables.

Q-Resolution. Given an assignment of the outermost existential variables, Goultiaeva et al. [GVGB11] presented a set of rules that reduces a Q-resolution proof under this assignment such that the result is again a Q-resolution proof. The values of the outermost universal variables in a winning strategy follow from the clause C in the reduced proof consisting only of universal literals—from such a clause the empty clause is derived by universal reduction. The polarity of the literal occurrences in C determines the value in the winning strategy. If a variable does not occur in C , or the conflict is derived by the application of the resolution rule, the value may be chosen freely. The approach works also for long-distance Q-resolution proofs [ELW13] and also for proofs [PSS16, PSS18] for reflexive resolution path.

Example 23. Consider the QBF $\exists x \forall a \exists z ((x \vee a \vee y) \wedge (\bar{x} \vee \bar{a} \vee y) \wedge \bar{y})$. If x is set to true, the approach presented in [GVGB11] simplifies the Q-resolution proof of the original formula to a Q-resolution proof for the formula $\forall a \exists z ((\bar{a} \vee y) \wedge \bar{y})$ that first resolves the two clauses and obtains the clause \bar{a} from which the empty clause is derived with universal reduction. Clause \bar{a} contains only universal literals, therefore, it is used to determine the value of a . Setting a to true results in the empty clause. This is the only choice for a winning move for the universal player under the assignment that sets x to true. If x had been set to false, the approach would force a to be set to false as well.

$\forall\text{Exp}+\text{Res}$. After the outermost existential variables have been assigned a truth value, the proof reduced under this assignment contains only annotations of the variables occurring in the outermost universal quantifier block in one polarity, i.e., those annotations are unopposed [BCJ14]. The polarities indicate how the respective universal variables have to be set in a winning strategy. Then the annotations of the outermost universal variables are removed, and the variables in the next existential block are assigned a value.

Example 24. Consider the QBF $\exists x \forall a \exists z ((x \vee a \vee y) \wedge (\bar{x} \vee \bar{a} \vee y) \wedge \bar{y})$. Instantiation results in the propositional formula $(x \vee y^0) \wedge (\bar{x} \vee y^1) \wedge \bar{y}^0 \wedge \bar{y}^1$ that is refuted in three steps, e.g., by resolving the first and the second clause resulting in the clause $(y^0 \vee y^1)$. Two further applications of the resolution rule result in the derivation

of the empty clause. If x is set to true, the instantiated formula simplifies to $(y^1) \wedge \bar{y}^0 \wedge \bar{y}^1$. The resolution proof simplifies respectively, and contains only unopposed annotations with a . Hence, a has to be set to true. The approach works similar for the case that x is set to false.

31.7. Connections Between Proof Systems

In this section we turn towards a proof-complexity analysis of the QBF proof systems defined earlier. Proof complexity offers two main threads towards this goal: a relative comparison of proof calculi via simulations and absolute (mostly exponential) lower bounds for the proof size of specific families of formulas in the systems. We survey and sketch some results for both these threads. Both approaches provide relevant information for solving as solver running time corresponds to proof size in the underlying calculi.

31.7.1. Enhanced Expansion Calculi

Before we start the comparison between the different calculi, we introduce two more proof systems that combine the CDCL and expansion calculi from Sections 31.3 and 31.4. Both these systems use the idea of expanding universal variables as in $\forall\text{Exp}+\text{Res}$ and keep track of the expansions in annotations, but incorporate additional features of CDCL-systems as in Section 31.3.

In $\forall\text{Exp}+\text{Res}$ each axiom clause is immediately annotated with a *fixed, complete* assignment to the universal variables. The proof then proceeds exactly as a propositional resolution proof, with clauses in fully annotated variables. In short, $\forall\text{Exp}+\text{Res}$ is propositional resolution on the conjuncts of a PCNF's complete expansion.

IR-calc, defined in [BCJ14], improves on this approach by working instead with *partial* assignments. In addition to resolution, the calculus is equipped with an *instantiation* rule by which partial annotations are grown throughout the course of the proof. To facilitate instantiation, the \circ operator describes how two partial assignments τ and σ are combined: $\tau \circ \sigma := \tau \cup \{l \in \sigma \mid \bar{l} \notin \tau\}$.

The rules of IR-calc are given in Figure 31.12. We provide a description of each of the IR-calc rules and illustrate them with some simple examples.

Axiom clauses are introduced into the proof, or *downloaded*, by selecting a clause C from the matrix and applying the *download assignment* to the existential literals. By design, the download assignment σ for C is the smallest partial assignment that falsifies every universal literal in C . Represented as a set of literals, then, $\sigma = \{\bar{l} \mid l \text{ is universal in } C\}$. When applying the download assignment, existentials are annotated only with universals *to their left* (i.e., those on which they depend). Consider the PCNF with prefix $\forall u \exists x \forall v \exists y$ and matrix clauses $\{u, \bar{x}, \bar{v}, y\}$ and $\{\bar{u}, x, \bar{y}\}$. Downloading the two matrix clauses gives rise to axioms $\{\bar{x}^{\bar{u}}, y^{\bar{u}v}\}$ and $\{x^u, \bar{y}^u\}$.

Instantiation applies a single partial assignment τ to all the annotations in a clause. As in the axiom rule, universals to the right of a variable are omitted from its annotation. Formally, for a partial assignment τ and an annotated clause

$$\frac{}{\{l^{[\tau]} \mid l \in C, l \text{ is an existential literal}\}} \text{ (Axiom)}$$

C is a non-tautological clause from the original matrix. τ is a partial 0/1 assignment to universal variables, which has all universal variables from C as its domain and falsifies all universal literals in C , i.e., $\tau = \{0/u \mid u \text{ is universal in } C\}$, where the notation $0/u$ for literals u is shorthand for $0/x$ if $u = x$ and $1/x$ if $u = \bar{x}$.

$$\frac{\{x^\tau\} \cup C_1 \quad \{\bar{x}^\tau\} \cup C_2}{C_1 \cup C_2} \text{ (Resolution)} \qquad \frac{C}{\text{inst}(\tau, C)} \text{ (Instantiation)}$$

C_1 , C_2 , and C are clauses of annotated variables.

τ is a partial 0/1 assignment to universal variables with $\text{rng}(\tau) \subseteq \{0, 1\}$.

Figure 31.12. The rules of IR-calc [BCJ14].

Axiom and instantiation rules as in IR-calc in Figure 31.12.

$$\frac{\{x^{\tau \cup \xi}\} \cup C_1 \quad \{\bar{x}^{\tau \cup \sigma}\} \cup C_2}{\text{inst}(\sigma, C_1) \cup \text{inst}(\xi, C_2)} \text{ (Resolution)}$$

C_1 and C_2 are clauses of annotated variables. τ is a partial assignment to universal variables (i.e., $\text{rng}(\tau) = \{0, 1\}$) and ξ, σ are extended partial assignments that assign universal variables to $\{0, 1, *\}$. $\text{dom}(\xi) \cap \text{dom}(\sigma) = \emptyset$.

$$\frac{C \cup \{l^\mu\} \cup \{l^\sigma\}}{C \cup \{l^\xi\}} \text{ (Merging)}$$

C is a clause of annotated variables. l is a literal and μ and σ are extended assignments with $\text{dom}(\mu) = \text{dom}(\sigma)$. $\xi = \{c/u \mid c/u \in \mu, c/u \in \sigma\} \cup \{*/u \mid c/u \in \mu, d/u \in \sigma, c \neq d\}$

Figure 31.13. The rules of IRM-calc [BCJ14]

C , the function $\text{inst}(\tau, C)$ returns the annotated clause $\{l^{[\sigma \circ \tau]} \mid l^\sigma \in C\}$. For example, given the prefix $\forall u_1 u_2 \exists x \forall u_3 u_4 \exists y$ and a clause $\{x^{u_1}, y^{\bar{u}_4}\}$, instantiation by $\tau = \bar{u}_2 u_3 u_4$ derives $\{x^{u_1 \bar{u}_2}, y^{u_3 \bar{u}_4}\}$. Note that u_3 and u_4 , which are right of x , do not appear in that variable's annotation after the instantiation. Also note that u_4 does not appear in the annotation to y , which is already annotated with the negated literal \bar{u}_4 before the instantiation takes place (see the earlier definition of \circ).

Resolution in IR-calc is identical to propositional resolution. We emphasize that annotations are labelling distinct variables (e.g., x^u and x^v are different variables), so that a resolution step is only valid if the annotations of the pivot literals match.

The calculus IRM-calc additionally incorporates long-distance steps in the spirit of LD-Q-Res. The rules of the calculus are depicted in Figure 31.13.

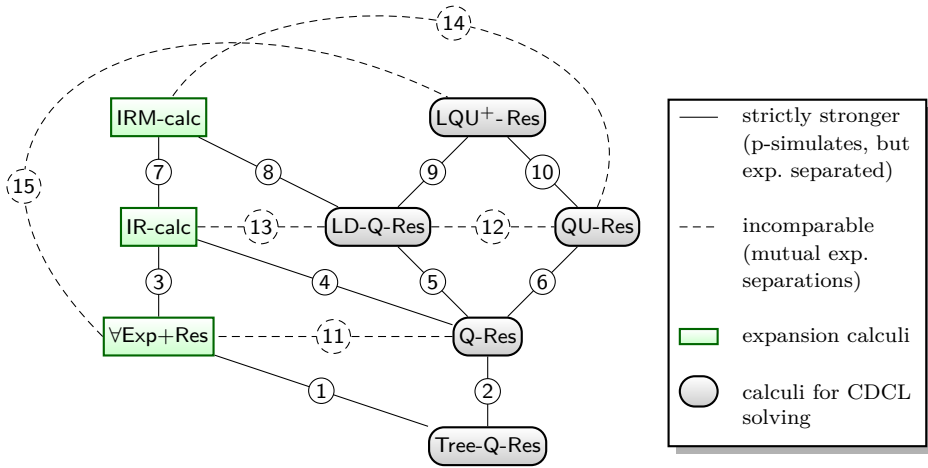


Figure 31.14. The simulation order of QBF resolution systems

31.7.2. The Simulation Order of QBF Resolution Calculi

We now provide an overview of the simulation order of the resolution systems defined earlier. A pictorial representation of the simulation order is given in Figure 31.14.

In Figure 31.14, solid lines denote simulations where the simulating system is placed at the higher position, e.g. $\forall\text{Exp}+\text{Res}$ simulates tree-like Q-Res by the simulation labeled with 1. Furthermore, all simulations are strict, e.g. tree-like Q-Res does not simulate $\forall\text{Exp}+\text{Res}$ as witnessed by a formula family with short proofs in $\forall\text{Exp}+\text{Res}$, but requiring exponential-size proofs in tree-like Q-Res. Dashed lines indicate incomparability results, e.g. $\forall\text{Exp}+\text{Res}$ and Q-Res are incomparable as per line 11. This means that there are formulas with polynomial-size proofs in $\forall\text{Exp}+\text{Res}$, but requiring exponential-size proofs in Q-Res, and vice versa.

We highlight that our information with respect to the simulation order of the eight depicted systems is complete, i.e., we either have a simulation result or an incomparability result. Missing lines follow by transitivity. We also point out that all the separations (either by strict simulations or incomparability) are exponential separations. References to the original results as well as pointers to the separating formulas are placed in Table 31.1 accompanying Figure 31.14.

Most of the simulation results follow straightforwardly from the definitions. This applies to the two simulations 3 and 7 between the expansion calculi on the left as well as to all simulations between the CDCL-type calculi on the right. Nontrivial simulations arise between conceptually different proof systems, which applies to simulations 1, 4 and 8, details of which can be found in [BCJ14].

Table 31.1. Formulas and references establishing simulations and separations. Numbers in the first column refer to the labels in Figure 31.14.

	Simulation	Separation
1	[JM15, BCMS18a]	[JM15], $QParity_n$ [BCCM18, BCJ15]
2	by def.	propositional formulas [BEGJ00]
3	[BCJ14]	[JM15, BCJ15]
4	[BCJ14]	$QParity_n$ [BCJ15]
5	by def.	$KBKF_n$ [ELW13], EQ_n [BBM19, BBH18a], $QParity_n$ [BCJ15, Che17b]
6	by def.	$KBKF_n$ [VG12a]
7	by def.	$KBKF_n$ [BCJ15, BB18], EQ_n [BBM19, BB18]
8	[BCJ14]	version of $QParity_n$ [BCJ15]
9	by def.	version of $KBKF_n$ [BWJ14]
10	by def.	version of $KBKF_n$ [BWJ14], EQ_n [BBH18a]
Incomparable		
11	[JM15], $QParity_n$ [BCJ15]	
12	versions of $KBKF_n$ [BWJ14]	
13	version of $QParity_n$ [BCJ15], $KBKF_n$ [BCJ15, BB18, ELW13]	
14	$QParity_n$ [BCJ15], version of $KBKF_n$ [BCJ15]	
15	version of $QParity_n$ [BCJ15], $KBKF_n$ [BCJ15]	

31.7.3. Lower Bounds and Separations

We now turn to the separations of the proof systems in Figure 31.14. This involves constructing specific formula families that provide the separations, e.g. for the separation of LD-Q-Res from Q-Res we need formulas ϕ_n that have polynomial-size proofs in LD-Q-Res, but require exponential-size proofs in Q-Res. For this separation we have multiple canonical choices for ϕ_n as either the equality formulas EQ_n from [BBH18a], the formulas $KBKF_n$ of Kleine Büning et al. [KBKF95], or the parity formulas [BCJ15].

In most instances, giving the upper bounds, i.e., constructing short proofs in the stronger system, turns out to be fairly easy, whereas establishing the lower bounds, i.e., showing the absence of short proofs in the weaker system, presents the main technical challenge. This is in line with the general picture in computational complexity, where the task of showing lower bounds turns out to be the most intricate.

Arguably, what is even more important than the actual lower bounds, is to devise generally applicable *lower-bound techniques*. For propositional resolution we have a number of such techniques (cf. [Seg07, Bus12]) and it is illuminating to review their applicability in the QBF context.

The (in)effectiveness of propositional techniques in QBF. The most widely used classical technique is the seminal *size-width technique* of [BSW01], which shows lower bounds for size via lower bounds for the width in resolution. However, this technique drastically fails even in the base system Q-Res [BCMS18a] and also does not seem applicable more widely [CB18].

Feasible interpolation is another widely-used propositional technique that connects circuit complexity to proof complexity. Feasible interpolation applies to a number of classical proof systems [Kra97, Pud97], and in [BCMS17] it was

demonstrated that this technique lifts to all QBF resolution systems depicted in Figure 31.14. However, the applicability of feasible interpolation is restricted by the fact that it imports lower bounds for monotone Boolean circuits (of which we only have relatively few [Juk12]).

A further general approach is through *Prover-Delayer games*. While this is effective in both propositional resolution [PI00, BGL13] as well as in QBF resolution [BCS17] (and can even yield optimal bounds [BGL13, BGL10, BCS17]), it only applies to the weak tree-like versions of these systems, where proofs are always in form of a tree, i.e., if derived formulas are needed multiple times in the proof they need to be rederived.

There is another short-coming of using propositional ideas in QBF. On existentially quantified formulas, all the QBF resolution systems coincide with classical proposition resolution. As such, all formulas hard for propositional resolution will be hard for any of the QBF resolution systems shown in Figure 31.14, e.g. the existentially quantified pigeonhole formulas are hard for all the QBF resolution calculi. Neither is this the phenomenon one wants to study in QBF proof complexity (cf. [Che17a, BHP17] for discussions), nor can such formulas provide any of the separations in Figure 31.14 (except for Q-Res from tree-like Q-Res). The same applies to the lifted interpolation technique: while it applies to new classes of QBFs (not just Σ_1^b formulas) it also does not provide any separations, as lower bounds via this technique will hold for all QBF resolution calculi simultaneously.

Summarising the current impact of propositional techniques in QBF it is fair to say that, at present, *ideas from the propositional world have limited impact to the QBF framework*, and *genuinely new ideas are needed* to cope with the more complex setting of quantifiers.

31.7.4. Genuine QBF Lower Bound Techniques

We will now turn to ideas for lower bounds specific to QBF and along the way explain some of the separations in Figure 31.14.

The principal approach to lower bounds in QBF is via *strategy extraction*, which originates from the game semantics of QBF. We recall the *two-player game* between an existential and a universal player, who in turn choose 0/1 values for the variables in the QBF in the order of the quantifier prefix, starting from the leftmost quantifier. The universal player wins as soon as one of the clauses in the QBF matrix gets falsified, otherwise the existential player wins. A fully quantified QBF is false if and only if the universal player has a winning strategy; likewise it is true if and only if the existential player has a winning strategy.

Strategy extraction computes a strategy from a proof of the formula. In particular, from a refutation of a false QBF in a QBF resolution system, a winning strategy for the universal player can be efficiently extracted [BCJ14, BJJW15]. This is practically important, as strategies witness the answers of QBF solvers [BJ12], but it can also be exploited for lower-bound techniques [BCJ15, BBC16, BBH18a].

The basic idea of this method is both conceptually simple and elegant: If we know that a family φ_n of false QBFs requires ‘complex’ winning strategies, then proofs of φ_n must be large in all proof systems with ‘feasible’ strategy extraction.

Lower bounds from circuit complexity. The qualification ‘complex’ can be given different specific definitions. The first is to consider the computational complexity required to compute the witnessing functions. For Q-Res and QU-Res we know that winning strategies can be computed by decision lists (cf. Section 31.6). It is easy to verify that decision lists can be turned into AC^0 circuits (cf. [BCJ15]). Hence winning strategies can be computed from Q-Res refutations in AC^0 .

To turn this into a lower bound, we need to construct false QBFs where all winning strategies are hard for bounded-depth circuits. As an example, consider the parity formulas $QParity_n$ from [BCJ15]

$$\exists x_1, \dots, x_n \forall z \exists t_2, \dots, t_n \text{ xor}(x_1, x_2, t_2) \cup \bigcup_{i=3}^n \text{ xor}(t_{i-1}, x_i, t_i) \cup \{z \vee t_n, \bar{z} \vee \bar{t}_n\}$$

where $\text{xor}(o_1, o_2, o)$ defines o to be $o_1 \oplus o_2$ through the following set of clauses

$$\{\bar{o}_1 \vee \bar{o}_2 \vee \bar{o}, o_1 \vee o_2 \vee \bar{o}, \bar{o}_1 \vee o_2 \vee o, o_1 \vee \bar{o}_2 \vee o\}.$$

Informally, $QParity_n$ reasons about $x_1 \oplus \dots \oplus x_n$, where the variables t_i encode the prefix sums $x_1 \oplus \dots \oplus x_i$. Hence t_n encodes $x_1 \oplus \dots \oplus x_n$. Through the only universal variable z the formula expresses the obvious contradiction that $x_1 \oplus \dots \oplus x_n$ is neither 0 nor 1. Crucially, the only strategy of the universal player to win on this false QBF is to play $z = x_1 \oplus \dots \oplus x_n$.

A seminal result of [FSS84, Hås87] states that every non-uniform family of bounded-depth circuits computing the parity function is of exponential size. Combined with the fact that strategy extraction for Q-Res and also QU-Res is possible in AC^0 , this results in an exponential lower bound for $QParity_n$ in Q-Res and QU-Res, first shown in [BCJ15].

In contrast, it turns out that the $QParity_n$ formulas admit short proofs both in $\forall\text{Exp}+\text{Res}$ [BCJ15] and in LD-Q-Res [Che17b]. We sketch the construction of the linear-size refutations of $QParity_n$ in $\forall\text{Exp}+\text{Res}$. We first instantiate all clauses in both polarities of z , generating the clauses

$$\text{xor}(x_1, x_2, t_2^{0/z}) \cup \bigcup_{i=3}^n \text{ xor}(t_{i-1}^{0/z}, x_i, t_i^{0/z}) \cup \{t_n^{0/z}\}$$

and $\text{xor}(x_1, x_2, t_2^{1/z}) \cup \bigcup_{i=3}^n \text{ xor}(t_{i-1}^{1/z}, x_i, t_i^{1/z}) \cup \{\bar{t}_n^{1/z}\}$.

Inductively, for $i = 2, \dots, n$ we now derive clauses representing $t_i^{0/z} \leftrightarrow t_i^{1/z}$. This yields a contradiction using the clauses $t_n^{0/z}$ and $\bar{t}_n^{1/z}$.

Therefore the $QParity$ formulas give one part of the separation between $\forall\text{Exp}+\text{Res}$ and Q-Res: formulas easy for $\forall\text{Exp}+\text{Res}$, but hard for Q-Res. Formulas easy in Q-Res, but hard in $\forall\text{Exp}+\text{Res}$ are contained in [JM15]. These formulas use n universal variables, which all need to be expanded in both polarities 0/1 in order to obtain an unsatisfiable set of clauses. Hence the expansion phase in all $\forall\text{Exp}+\text{Res}$ refutations of the formula is of size 2^n . It is not difficult to construct such formulas that are also easy for Q-Res.

Lower bounds through cost. We now explore a second way of how to interpret the qualification of ‘complex’ in winning strategies. Following the semantic approach of [BBH18a], we consider false QBFs where all winning strategies require

many moves of the universal player. We will see that such formulas are hard for QU-Res and further QBF systems.

We measure the size of winning strategies (for a single block) by the cost of a formula.

Definition 31.7.1 ([BBH18a]). The *cost* of a false QBF is the minimum, over all winning strategies, of the largest number of responses for a single universal block.

Strategies that require many responses of the universal player (in one block) are costly. This measure provides an absolute lower bound on the number of clauses in QU-Res refutations.

Theorem 31.7.1 ([BBH18a]). *Let π be a QU-Res refutation of a QBF Φ . Then $|\pi| \geq \text{cost}(\Phi)$.*

We consider the *equality formulas* from [BBH18a] as an example. The QBF EQ_n is defined as

$$\exists x_1 \cdots x_n \forall u_1 \cdots u_n \exists t_1 \cdots t_n \left(\bigwedge_{i=1}^n (x_i \vee u_i \vee \bar{t}_i) \wedge (\bar{x}_i \vee \bar{u}_i \vee \bar{t}_i) \right) \wedge \left(\bigvee_{i=1}^n t_i \right).$$

The only winning strategy for these formulas is to play $u_i = x_i$ for $i = 1, \dots, n$. There is only one universal block. Hence the cost of EQ_n is 2^n and all QU-Res (and Q-Res) proofs of EQ_n are of exponential size.

In contrast, EQ_n are easily verified to have linear-size proofs in LD-Q-Res [BBM19], hence the formulas provide the separation between Q-Res and LD-Q-Res (line 5 in Figure 31.14).

Another such separation is provided by the $KBKF_n$ formulas of Kleine Büning et al. [KBKF95], which appear widely in the QBF literature (for examples, see [Egl16, BCJ15, BWJ14, LES16]). Again these formulas can be shown to be hard for Q-Res via a cost argument [BBH18a], while they are easy for QU-Res [VG12a] and LD-Q-Res [ELW13], thus in particular providing the separation between QU-Res and Q-Res. Modified versions of the $KBKF_n$ formulas also provide the separations of LQU⁺-Res from LD-Q-Res and QU-Res [BWJ14].

A related notion of cost is also effective as a lower bound measure for expansion calculi, whereby the $KBKF_n$ formulas can be shown to be hard for IR-calc [BB18].

31.7.5. Relations to Further Calculi

We briefly mention some facts on the proof complexity of further approaches defined in earlier subsections: dependency schemes, symmetries, and QRAT. However, in comparison to the systems in Figure 31.14 for which we know their precise relations, we only have partial results on their relative complexity.

For *dependency schemes* (cf. Section 31.3.2), the EQ_n and $KBKF_n$ formulas give theoretical justification for their use: both formulas are hard for Q-Res, but become easy when using the resolution path dependency scheme [BB17]. In fact, even the standard dependency scheme cannot identify any useful dependencies

for the EQ_n and $KBKF_n$ formulas, i.e., the formulas exponentially separate Q-Res with the resolution path dependency scheme from Q-Res with the standard dependency scheme.

In addition, the formulas $KBKF_n$ and $QParity_n$ have short proofs if Q-Res is extended by the *symmetry rule* (see Section 31.3.2). It is, however, easy to destroy the symmetries by introducing universal pure literals into those formulas such that no symmetries can be exploited. Hence LQU^+ -Res and Q-Res with the symmetry rule are incomparable.

For the QRAT proof system (see Section 31.5) it is known that it is strictly stronger than LD-Q-Res and QU-Res [KHS17]. Currently nothing is known about the relationships to LQU^+ -Res or the expansion calculi.

31.7.6. Stronger QBF Proof Systems

We now turn briefly to the wider landscape of stronger QBF calculi beyond resolution. We have seen previously that propositional resolution can be turned into the QBF system QU-Res by simply adding the universal reduction rule. This procedure can be generalised [BBC16]: every line-based propositional proof system P (fulfilling some natural conditions [BBH18a]) can be lifted to a system $P+\forall\text{red}$, which is sound and complete for QBF.

In this way, we obtain natural QBF versions of the geometric system *cutting planes* [BCMS18b], working with linear inequalities instead of clauses, of the algebraic system *polynomial calculus*, working with polynomials, and of the logical *Frege systems*, working with arbitrary finite sets of axioms and a set of propositional rules [BBC16].

We note that the two lower bound techniques via strategy extraction from Section 31.7.4 remain applicable to these much stronger QBF systems. However, when using circuit complexity as in the first approach, we need to import stronger (conditional) circuit lower bounds, i.e., for NC^1 in case of Frege and for P/poly in case of extended Frege [BBC16].

For the QBF Frege systems, we can even *characterise* their reasons for hardness: lower bounds either arise through a lower bound for propositional Frege or through a circuit lower bound (the precise condition is $NC^1 \not\subseteq PSPACE$ in case of Frege) [BP16]. Hence, QBF Frege systems naturally unite the hardest problem in propositional proof complexity (lower bounds for Frege) with the hardest task in circuit complexity (unconditional circuit lower bounds).

The second approach via cost is also effective for strong systems. However, here we need to include another measure: the *capacity* of proof lines. Informally, the capacity of a line in a proof counts how many responses can be at most extracted from it; and the capacity of a proof is defined as the maximum of the capacities of its lines. As an example, clauses have capacity one as the universal player has a unique strategy to falsify the clause. Hence resolution proofs always have capacity one.

We recall that cost measures how many responses need to be extracted from the proof and that strategy extraction is efficient by the round-based algorithm in Section 31.6. This leads to the *size-cost-capacity theorem* [BBH18a], a general result for QBF proof systems of the form $P+\forall\text{red}$, stating that for each QBF Φ

and each $P+\forall$ red proof π of Φ

$$|\pi| \geq \frac{\text{cost}(\Phi)}{\text{capacity}(\pi)} .$$

By this theorem, the equality formulas are hard not only for QU-Res, but also in QBF cutting planes and polynomial calculus [BBH18a]. Via this approach, we can also show the hardness of a large class of random QBFs in these calculi.

In contrast, QBF Frege admits short proofs for equality and these random formulas, as the capacity of Frege lines can become exponentially large.

Finally, we also mention *sequent calculi* for QBF [KP90, CM05]. In contrast to all the systems considered before they can also manipulate the quantifier prefix and thus are very strong systems, even stronger than QBF Frege [BP16]. However, there are also various restricted versions of intermediate strength [Egl12].

QBF sequent calculi and Frege systems enjoy a close connection to first-order theories of bounded arithmetic [KP90, CM05, BP16]. Via different translations many of the QBF resolution systems considered here are also closely related to first-order logic [Egl16, SLB12].

References

- [AB02] A. Ayari and D. A. Basin. QUBOS: Deciding Quantified Boolean Logic Using Propositional Satisfiability Solvers. In *FMCAD*, volume 2517 of *LNCS*, pages 187–201. Springer, 2002.
- [BB16] O. Beyersdorff and J. Blinkhorn. Dependency Schemes in QBF Calculi: Semantics and Soundness. In *CP*, volume 9892 of *LNCS*, pages 96–112. Springer, 2016.
- [BB17] J. Blinkhorn and O. Beyersdorff. Shortening QBF Proofs with Dependency Schemes. In *SAT*, volume 10491 of *LNCS*, pages 263–280. Springer, 2017.
- [BB18] O. Beyersdorff and J. Blinkhorn. Genuine lower bounds for QBF expansion. In *STACS*, volume 96 of *LIPICs*, pages 12:1–12:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.
- [BBC16] O. Beyersdorff, I. Bonacina, and L. Chew. Lower Bounds: From Circuits to QBF Proof Systems. In *ITCS*, pages 249–260. ACM, 2016.
- [BBH18a] O. Beyersdorff, J. Blinkhorn, and L. Hinde. Size, cost, and capacity: A semantic technique for hard random QBFs. In *ITCS*, pages 9:1–9:18. LIPICs, 2018.
- [BBH⁺18b] R. Bloem, N. Braud-Santoni, V. Hadzic, U. Egly, F. Lonsing, and M. Seidl. Expansion-based QBF solving without recursion. In *FM-CAD*, pages 1–10. IEEE, 2018.
- [BBM19] O. Beyersdorff, J. Blinkhorn, and M. Mahajan. Building strategies into QBF proofs. In *STACS*, to appear in LIPICs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019.
- [BCCM18] O. Beyersdorff, L. Chew, J. Clymo, and M. Mahajan. Short proofs in QBF expansion. *ECCC*, 25:102, 2018.

- [BCJ14] O. Beyersdorff, L. Chew, and M. Janota. On unification of QBF resolution-based calculi. In *MFCS, II*, pages 81–93, 2014.
- [BCJ15] O. Beyersdorff, L. Chew, and M. Janota. Proof complexity of resolution-based QBF calculi. In *STACS*, volume 30 of *LIPICs*, pages 76–89. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.
- [BCMS17] O. Beyersdorff, L. Chew, M. Mahajan, and A. Shukla. Feasible interpolation for QBF resolution calculi. *Logical Methods in Computer Science*, 13, 2017.
- [BCMS18a] O. Beyersdorff, L. Chew, M. Mahajan, and A. Shukla. Are short proofs narrow? QBF resolution is not so simple. *ACM Transactions on Computational Logic*, 19, 2018.
- [BCMS18b] O. Beyersdorff, L. Chew, M. Mahajan, and A. Shukla. Understanding cutting planes for QBFs. *Inf. Comput.*, 262:141–161, 2018.
- [BCS17] O. Beyersdorff, L. Chew, and K. Sreenivasaiiah. A game characterisation of tree-like Q-resolution size. *J. of Computer and System Sciences*, 2017. in press.
- [BEGJ00] M. L. Bonet, J. L. Esteban, N. Galesi, and J. Johannsen. On the relative complexity of resolution refinements and cutting planes proof systems. *SIAM J. Comput.*, 30(5):1462–1484, 2000.
- [Ben04] M. Benedetti. Evaluating qbfs via symbolic skolemization. In *LPAR*, volume 3452 of *LNCS*, pages 285–300. Springer, 2004.
- [Ben05] M. Benedetti. Quantifier Trees for QBFs. In *SAT*, volume 3569 of *LNCS*, pages 378–385. Springer, 2005.
- [BG01] L. Bachmair and H. Ganzinger. Resolution theorem proving. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 19–99. Elsevier and MIT Press, 2001.
- [BGL10] O. Beyersdorff, N. Galesi, and M. Lauria. A lower bound for the pigeonhole principle in tree-like resolution by asymmetric prover-delayer games. *Inf. Process. Lett.*, 110(23):1074–1077, 2010.
- [BGL13] O. Beyersdorff, N. Galesi, and M. Lauria. A characterization of tree-like resolution size. *Inf. Process. Lett.*, 113(18):666–671, 2013.
- [BHP17] O. Beyersdorff, L. Hinde, and J. Pich. Reasons for hardness in QBF proof systems. In *FSTTCS*, pages 14:1–14:15, 2017.
- [Bie04] A. Biere. Resolve and expand. In *SAT (Selected Papers)*, volume 3542 of *LNCS*, pages 59–70. Springer, 2004.
- [BJ11] V. Balabanov and J. R. Jiang. Resolution Proofs and Skolem Functions in QBF Evaluation and Applications. In *CAV*, volume 6806 of *LNCS*, pages 149–164. Springer, 2011.
- [BJ12] V. Balabanov and J. R. Jiang. Unified QBF Certification and its Applications. *Formal Methods in System Design*, 41(1):45–65, 2012.
- [BJJW15] V. Balabanov, J. R. Jiang, M. Janota, and M. Widl. Efficient Extraction of QBF (Counter)models from Long-Distance Resolution Proofs. In *AAAI*, pages 3694–3701. AAAI Press, 2015.
- [BK07] U. Bubeck and H. Kleine Büning. Bounded Universal Expansion for Preprocessing QBF. In *SAT*, volume 4501 of *LNCS*, pages 244–257. Springer, 2007.

- [BLS11] A. Biere, F. Lonsing, and M. Seidl. Blocked Clause Elimination for QBF. In *CADE*, volume 6803 of *LNCS*, pages 101–115. Springer, 2011.
- [BP16] O. Beyersdorff and J. Pich. Understanding Gentzen and Frege systems for QBF. In *LICS*, pages 146–155, 2016.
- [BSW01] E. Ben-Sasson and A. Wigderson. Short proofs are narrow - resolution made simple. *J. of the ACM*, 48(2):149–169, 2001.
- [Bus12] S. R. Buss. Towards NP-P via proof complexity and search. *Ann. Pure Appl. Logic*, 163(7):906–917, 2012.
- [BWJ14] V. Balabanov, M. Widl, and J. R. Jiang. QBF Resolution Systems and Their Proof Complexities. In *SAT*, volume 8561 of *LNCS*, pages 154–169. Springer, 2014.
- [CB18] J. Clymo and O. Beyersdorff. Relating size and width in variants of Q-resolution. *Inf. Process. Lett.*, 138:1–6, 2018.
- [CGJ+03] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
- [CGS98a] M. Cadoli, A. Giovanardi, and M. Schaerf. An Algorithm to Evaluate Quantified Boolean Formulae. In *AAAI/IAAI*, pages 262–267. AAAI Press / The MIT Press, 1998.
- [CGS98b] M. Cadoli, A. Giovanardi, and M. Schaerf. An Algorithm to Evaluate Quantified Boolean Formulae. In *AAAI*, pages 262–267. AAAI Press / The MIT Press, 1998.
- [Che17a] H. Chen. Proof complexity modulo the polynomial hierarchy: Understanding alternation as a source of hardness. *TOCT*, 9(3):15:1–15:20, 2017.
- [Che17b] L. Chew. *QBF proof complexity*. PhD thesis, University of Leeds, 2017.
- [CM05] S. A. Cook and T. Morioka. Quantified propositional calculus and a second-order theory for NC^1 . *Arch. Math. Log.*, 44(6):711–749, 2005.
- [Egl12] U. Egly. On sequent systems and resolution for QBFs. In *SAT*, volume 7317 of *LNCS*, pages 100–113. Springer, 2012.
- [Egl16] U. Egly. On stronger calculi for QBFs. In *SAT*, volume 9710 of *LNCS*, pages 419–434. Springer, 2016.
- [ELW13] U. Egly, F. Lonsing, and M. Widl. Long-Distance Resolution: Proof Generation and Strategy Extraction in Search-Based QBF Solving. In *LPAR*, volume 8312 of *LNCS*, pages 291–308. Springer, 2013.
- [FHSB17] K. Fazekas, M. J. H. Heule, M. Seidl, and A. Biere. Skolem Function Continuation for Quantified Boolean Formulas. In *TAP*, volume 10375 of *LNCS*, pages 129–138. Springer, 2017.
- [FSS84] M. L. Furst, J. B. Saxe, and M. Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory*, 17(1):13–27, 1984.
- [GB13] A. Goultiaeva and F. Bacchus. Recovering and utilizing partial duality in QBF. In *SAT*, volume 7962 of *LNCS*, pages 83–99. Springer, 2013.
- [GMN10] E. Giunchiglia, P. Marin, and M. Narizzano. sQueueBF: An Effective

- Preprocessor for QBFs Based on Equivalence Reasoning. In *SAT*, volume 6175 of *LNCS*, pages 85–98. Springer, 2010.
- [GNT06] E. Giunchiglia, M. Narizzano, and A. Tacchella. Clause/Term Resolution and Learning in the Evaluation of Quantified Boolean Formulas. *JAIR*, 26:371–416, 2006.
- [GNT07] E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantifier Structure in Search-Based Procedures for QBFs. *TCAD*, 26(3):497–507, 2007.
- [GVGB11] A. Goultiaeva, A. Van Gelder, and F. Bacchus. A Uniform Approach for Generating Proofs and Strategies for Both True and False QBF Formulas. In *IJCAI*, pages 546–553. IJCAI/AAAI, 2011.
- [Hås87] J. Håstad. *Computational Limitations of Small Depth Circuits*. MIT Press, Cambridge, 1987.
- [HJL⁺15] M. J. H. Heule, M. Järvisalo, F. Lonsing, M. Seidl, and A. Biere. Clause Elimination for SAT and QSAT. *J. Artif. Intell. Res.*, 53:127–168, 2015.
- [HKB17] M. J. H. Heule, B. Kiesl, and A. Biere. Short Proofs Without New Variables. In *CADE*, volume 10395 of *LNCS*, pages 130–147. Springer, 2017.
- [HSB14] M. J. H. Heule, M. Seidl, and A. Biere. Efficient extraction of Skolem functions from QRAT proofs. In *FMCAD*, pages 107–114. IEEE, 2014.
- [HSB15] M. J. H. Heule, M. Seidl, and A. Biere. Blocked Literals Are Universal. In *NFM 2015*, volume 9058 of *LNCS*, pages 436–442. Springer, 2015.
- [HSB17] M. J. H. Heule, M. Seidl, and A. Biere. Solution validation and extraction for QBF preprocessing. *J. Autom. Reasoning*, 58(1):97–125, 2017.
- [Jan18a] M. Janota. Circuit-based search space pruning in QBF. In *SAT*, volume 10929 of *LNCS*, pages 187–198. Springer, 2018.
- [Jan18b] M. Janota. Towards generalization in QBF solving via machine learning. In *AAAI*, pages 6607–6614. AAAI Press, 2018.
- [JGM13] M. Janota, R. Grigore, and J. Marques-Silva. On QBF Proofs and Preprocessing. In *LPAR*, volume 8312 of *LNCS*, pages 473–489. Springer, 2013.
- [JJK⁺14] M. Janota, C. Jordan, W. Klieber, F. Lonsing, M. Seidl, and A. Van Gelder. The QBF Gallery 2014: The QBF competition at the FLoC Olympic games. *JSAT*, 9:187–206, 2014.
- [JKMC16] M. Janota, W. Klieber, J. Marques-Silva, and E. M. Clarke. Solving QBF with counterexample guided refinement. *Artif. Intell.*, 234:1–25, 2016.
- [JKMSC12] M. Janota, W. Klieber, J. Marques-Silva, and E. Clarke. Solving QBF with counterexample guided refinement. In *SAT*, volume 7317 of *LNCS*, pages 114–128. Springer, 2012.
- [JM15] M. Janota and J. Marques-Silva. Expansion-based QBF solving versus Q-resolution. *Theor. Comput. Sci.*, 577:25–42, 2015.
- [JMS11] M. Janota and J. Marques-Silva. Abstraction-based algorithm for 2QBF. In *SAT*, volume 6695 of *LNCS*, pages 230–244. Springer,

- 2011.
- [JMS13] M. Janota and J. Marques-Silva. On propositional QBF expansions and Q-resolution. In *SAT*, volume 7962 of *LNCS*, pages 67–82. Springer, 2013.
 - [Juk12] S. Jukna. *Boolean Function Complexity - Advances and Frontiers*, volume 27 of *Algorithms and combinatorics*. Springer, 2012.
 - [KBKF95] H. Kleine Büning, M. Karpinski, and A. Flögel. Resolution for Quantified Boolean Formulas. *Inf. Comput.*, 117(1):12–18, 1995.
 - [KHS17] B. Kiesl, M. J. H. Heule, and M. Seidl. A little blocked literal goes a long way. In *SAT*, volume 10491 of *Lecture Notes in Computer Science*, pages 281–297. Springer, 2017.
 - [KP90] J. Krajíček and P. Pudlák. Quantified propositional calculi and fragments of bounded arithmetic. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 36:29–46, 1990.
 - [Kra97] J. Krajíček. Interpolation theorems, lower bounds for proof systems and independence results for bounded arithmetic. *J. of Symbolic Logic*, 62(2):457–486, 1997.
 - [KS18a] M. Kauers and M. Seidl. Short proofs for some symmetric quantified boolean formulas. *Inf. Process. Lett.*, 140:4–7, 2018.
 - [KS18b] M. Kauers and M. Seidl. Symmetries of quantified boolean formulas. In *SAT*, volume 10929 of *LNCS*, pages 199–216. Springer, 2018.
 - [LB09] F. Lonsing and A. Biere. A Compact Representation for Syntactic Dependencies in QBFs. In *SAT*, volume 5584 of *LNCS*, pages 398–411. Springer, 2009.
 - [LB10] F. Lonsing and A. Biere. Integrating Dependency Schemes in Search-Based QBF Solvers. In *SAT*, volume 6175 of *LNCS*, pages 158–171. Springer, 2010.
 - [LBB⁺15] F. Lonsing, F. Bacchus, A. Biere, U. Egly, and M. Seidl. Enhancing Search-Based QBF Solving by Dynamic Blocked Clause Elimination. In *LPAR*, volume 9450 of *LNCS*, pages 418–433. Springer, 2015.
 - [LE17] F. Lonsing and U. Egly. DepQBF 6.0: A Search-Based QBF Solver Beyond Traditional QCDCL. In *CADE*, volume 10395 of *LNCS*, pages 371–384. Springer, 2017.
 - [LE18] F. Lonsing and U. Egly. QRAT⁺: Generalizing QRAT by a More Powerful QBF Redundancy Property. In *IJCAR*, volume 10900 of *LNCS*, pages 161–177. Springer, 2018.
 - [LES16] F. Lonsing, U. Egly, and M. Seidl. Q-Resolution with Generalized Axioms. In *SAT*, volume 9710 of *LNCS*, pages 435–452. Springer, 2016.
 - [Let02] R. Letz. Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas. In *TABLEAUX*, volume 2381 of *LNCS*, pages 160–175. Springer, 2002.
 - [LEVG13] F. Lonsing, U. Egly, and A. Van Gelder. Efficient clause learning for quantified boolean formulas via QBF pseudo unit propagation. In *SAT*, volume 7962 of *LNCS*, pages 100–115. Springer, 2013.
 - [LSVG16] F. Lonsing, M. Seidl, and A. Van Gelder. The QBF Gallery: Behind the Scenes. *Artif. Intell.*, 237:92–114, 2016.

- [MMZ⁺01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC*, pages 530–535. ACM, 2001.
- [MSS99] J. P. Marques Silva and K. A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.
- [PG86] D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *J. Symb. Comput.*, 2(3):293–304, 1986.
- [PI00] P. Pudlák and R. Impagliazzo. A lower bound for DLL algorithms for SAT. In *SODA*, pages 128–136, 2000.
- [PSS16] T. Peitl, F. Slivovsky, and S. Szeider. Long Distance Q-Resolution with Dependency Schemes. In *SAT*, volume 9710 of *LNCS*, pages 500–518. Springer, 2016.
- [PSS17] T. Peitl, F. Slivovsky, and S. Szeider. Dependency Learning for QBF. In *SAT*, volume 10491 of *LNCS*, pages 298–313. Springer, 2017.
- [PSS18] T. Peitl, F. Slivovsky, and S. Szeider. Long-Distance Q-Resolution with Dependency Schemes. *J. of Aut. Reas.*, 2018.
- [Pud97] P. Pudlák. Lower bounds for resolution and cutting planes proofs and monotone computations. *J. of Symbolic Logic*, 62(3):981–998, 1997.
- [PV04] G. Pan and M. Y. Vardi. Symbolic decision procedures for QBF. In *CP*, volume 3258 of *LNCS*, pages 453–467. Springer, 2004.
- [Rob65] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, 1965.
- [RT15] M. N. Rabe and L. Tentrup. CAQE: A certifying QBF solver. In *FMCAD*, pages 136–143. IEEE, 2015.
- [RTRS18] M. N. Rabe, L. Tentrup, C. Rasmussen, and S. A. Seshia. Understanding and extending incremental determinization for 2qbf. In *CAV (2)*, volume 10982 of *Lecture Notes in Computer Science*, pages 256–274. Springer, 2018.
- [Sam08] M. Samer. Variable Dependencies of Quantified CSPs. In *LPAR*, volume 5330 of *LNCS*, pages 512–527. Springer, 2008.
- [SDB06] H. Samulowitz, J. Davies, and F. Bacchus. Preprocessing QBF. In *CP*, volume 4204 of *LNCS*, pages 514–529. Springer, 2006.
- [Seg07] N. Segerlind. The complexity of propositional proofs. *Bulletin of Symbolic Logic*, 13(4):417–481, 2007.
- [SLB12] M. Seidl, F. Lonsing, and A. Biere. qbf2epr: A tool for generating EPR formulas from QBF. In *Third Workshop on Practical Aspects of Automated Reasoning, PAAR-2012, Manchester, UK, June 30 - July 1, 2012*, pages 139–148, 2012.
- [SS09] M. Samer and S. Szeider. Backdoor Sets of Quantified Boolean Formulas. *J. of Aut. Reas.*, 42(1):77–97, 2009.
- [SS12] F. Slivovsky and S. Szeider. Computing Resolution-Path Dependencies in Linear Time. In *SAT*, volume 7317 of *LNCS*, pages 58–71. Springer, 2012.
- [SS16] F. Slivovsky and S. Szeider. Soundness of Q-resolution with dependency schemes. *Theor. Comput. Sci.*, 612:83–101, 2016.

- [Ten16] L. Tentrup. Non-prenex QBF solving using abstraction. In *SAT*, volume 9710 of *LNCS*, pages 393–401. Springer, 2016.
- [THJ15] K. Tu, T. Hsu, and J. R. Jiang. QELL: QBF reasoning with extended clause learning and leveled SAT solving. In *SAT*, volume 9340 of *LNCS*, pages 343–359. Springer, 2015.
- [Tse68] G. S. Tseitin. On the complexity of derivations in the propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic*, 1968.
- [VG11] A. Van Gelder. Variable Independence and Resolution Paths for Quantified Boolean Formulas. In *CP*, volume 6876 of *LNCS*, pages 789–803. Springer, 2011.
- [VG12a] A. Van Gelder. Contributions to the Theory of Practical Quantified Boolean Formula Solving. In *CP*, volume 7514 of *LNCS*, pages 647–663. Springer, 2012.
- [VG12b] A. Van Gelder. Producing and verifying extremely large propositional refutations - have your cake and eat it too. *Ann. Math. Artif. Intell.*, 65(4):329–372, 2012.
- [VGWL12] A. Van Gelder, S. B. Wood, and F. Lonsing. Extended Failed-Literal Preprocessing for Quantified Boolean Formulas. In *SAT*, volume 7317 of *LNCS*, pages 86–99. Springer, 2012.
- [WHHJ14] N. Wetzler, M. J. H. Heule, and W. A. Hunt Jr. DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In *SAT*, volume 8561 of *LNCS*, pages 422–429. Springer, 2014.
- [WRMB17] R. Wimmer, S. Reimer, P. Marin, and B. Becker. HQSpre - An Effective Preprocessor for QBF and DQBF. In *TACAS*, volume 10205 of *LNCS*, pages 373–390, 2017.
- [ZM02a] L. Zhang and S. Malik. Conflict Driven Learning in a Quantified Boolean Satisfiability Solver. In *ICCAD*, pages 442–449. ACM / IEEE Computer Society, 2002.
- [ZM02b] L. Zhang and S. Malik. Towards a Symmetric Treatment of Satisfaction and Conflicts in Quantified Boolean Formula Evaluation. In *CP*, volume 2470 of *LNCS*, pages 200–215. Springer, 2002.
- [ZMMM01] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in Boolean Satisfiability Solver. In *ICCAD*, pages 279–285. IEEE, 2001.

This page intentionally left blank

Chapter 32

SAT Techniques for Modal and Description Logics

Roberto Sebastiani and Armando Tacchella

32.1. Introduction

In a nutshell, modal logics are propositional logics enriched with *modal operators*, —like \Box , \Diamond , \Box_i — which are able to represent complex facts like necessity, possibility, knowledge and belief. For instance, “ $\Box\varphi$ ” and “ $\Diamond\varphi$ ” may represent “necessarily φ ” and “possibly φ ” respectively, whilst “ $\Box_1\Box_2\varphi$ ” may represent the fact that agent 1 knows that agent 2 knows the fact φ . Description logics are extensions of propositional logic which build on top of entities, concepts (unary relations) and roles (binary relations), which allow for representing complex concepts. For instance, the concept “ $\text{MALE} \wedge \exists \text{CHILDREN} (\neg \text{MALE} \wedge \text{TEEN})$ ”, represents the set of fathers which have at least one teenager daughter.

The research in modal and description logics had followed two parallel routes until the seminal work by Schild [Sch91], who showed that the core modal logic K_m and the core description logic \mathcal{ALC} are notational variants one of the other, and that analogous frameworks, results and algorithms had been conceived in parallel in the two communities. Since then, analogous results have been produced for a bunch of other logics, so that nowadays the two communities have substantially merged into one research flow.

In the last two decades, modal and description logics have provided a theoretical framework for important applications in many areas of computer science, including artificial intelligence, formal verification, database theory, distributed computing and, more recently, semantic web. For this reason, the problem of automated reasoning in modal and description logics has been thoroughly investigated (see, e.g., [Fit83, Lad77, HM92, BH91, Mas00]), and many approaches have been proposed for (efficiently) handling the satisfiability of modal and description logics, with a particular interest for the core logics K_m and \mathcal{ALC} (see, e.g., [Fit83, BH91, GS00, HPS99, HS99, BGdR03, PSV02, SV06]). Moreover, a significant amount of benchmarks formulas have been produced for testing the effectiveness of the different techniques [HM92, GRS96, HS96, HPSS00, Mas99, PSS01, PSS03].

We briefly overview the main approaches for the satisfiability of modal and description logics which have been proposed in the literature. The “classic” *tableau-based* approach [Fit83, Lad77, HM92, Mas00] is based on the construction of propositional-tableau branches, which are recursively expanded on demand by generating successor nodes in a candidate Kripke model. In the *DPLL-based* approach [GS96a, SV98, GS00] a DPLL procedure, which treats the modal subformulas as propositions, is used as Boolean engine at each nesting level of the modal operators: when a satisfying assignment is found, the corresponding set of modal subformulas is recursively checked for modal consistency. Among the tools employing (and extending) this approach, we recall KSAT [GS96a, GGST00], *SAT [Tac99], FACT [Hor98b], DLP [PS98], and RACER [HM01].¹ This approach has lately been exported into the context of Satisfiability Modulo Theories - SMT [ACG00, ABC⁺02], giving rise the so-called *on-line lazy approach* to SMT described in §26.4 (see also [Seb07]). The *CSP-based* approach [BGdR03] differs from the tableaux-based and DPLL-based ones mostly in the fact that a CSP engine is used instead of a tableaux/DPLL engine. KCSP is the representative tool of this approach. In the *translational* approach [HS99, AGHd00] the modal formula is encoded into first-order logic (FOL), and the encoded formula is then fed to a FOL theorem prover [AGHd00]. MSPASS [HSW99] is the most representative tool of this approach. In the *Inverse-method* approach, a search procedure is based on the inverted version of a sequent calculus [Vor99, Vor01] (which can be seen as a modalized version of propositional resolution [PSV02]). K \mathcal{X} is the representative tool of this approach. In the *Automata-theoretic* approach, or *OBDD-based* approach, (a OBDD-based symbolic representation of) a tree automaton accepting all the tree models of the input formula is implicitly built and checked for emptiness [PSV02, PV03]. KBDD [PV03] is the representative tool of this approach. [PV03] presents also an encoding of K-satisfiability into QBF-satisfiability – another PSPACE-complete problem – combined with the use of a state-of-the-art QBF solver. Finally, in the *eager approach* [SV06, SV08] K_m /*ACC*-formulas are encoded into SAT and then fed to a state-of-the-art SAT solver. K_m2SAT is the representative tool of this approach.

Most such approaches combine propositional reasoning with various techniques/encodings for handling the modalities, and thus are based on, or have largely benefited from, efficient propositional reasoning techniques. In particular, the usage of DPLL as a core Boolean reasoning technique produced a boost in the performance of the tools when it was adopted [GS96a, GS96b, Hor98b, PS98, HPS99, GGST00, HPSS00].

In this chapter we show how efficient Boolean reasoning techniques have been imported, used and integrated into reasoning tools for modal and description logics. To this extent, we focus on modal logics, and in particular mainly on K_m . Importantly, this chapter *does not* address the much more general issue of satisfiability in modal and description logics, because the reasoning techniques

¹ Notice that there is not an universal agreement on the terminology “tableau-based” and “DPLL-based”. E.g., tools like FACT, DLP, and RACER are often called “tableau-based”, although they use a DPLL-like algorithm instead of propositional tableaux for handling the propositional component of reasoning [Hor98b, PS98, HPS99, HM01], because many scientists in these communities consider DPLL as an optimized version of propositional tableaux. The same issue holds for the Boolean system KE [DM94] and its derived systems.

which are specific for the different modal and description logics are orthogonal to the issue of Boolean reasoning. We refer the reader to the bibliography presented above and to [BCM⁺03] for a detailed description of those topics.

The chapter is organized as follows. In §32.2 we provide some background in modal logics. In §32.3 we describe a basic theoretical framework and we present and analyze the basic tableau-based and DPLL-based techniques. In §32.4 we present optimizations and extensions of the DPLL-based procedures. In §32.5 we present the automata-theoretic/OBDD-based approach. Finally, in §32.6 we present the eager approach.

32.2. Background

In this section we provide some background in modal logics. We refer the reader to, e.g., [Che80, Fit83, HM92] for a more detailed introduction.

32.2.1. The Modal Logic K_m

We start with some basic notions and notation (see, e.g., [Che80, Fit83, HM92] for more details). Given a non-empty set of primitive propositions $\mathcal{A} = \{A_1, A_2, \dots\}$ and a set of m modal operators $\mathcal{B} = \{\Box_1, \dots, \Box_m\}$, let the language Λ_m be the least set of formulas containing \mathcal{A} , closed under the set of propositional connectives $\{\neg, \wedge\}$ and the set of modal operators in \mathcal{B} . Notationally, we use capital letters A_i, B_i, \dots to denote primitive propositions and Greek letters $\alpha_i, \beta_i, \varphi_i, \psi_i$ to denote formulas in Λ_m . We use the standard abbreviations, that is: “ $\diamond_r \varphi_1$ ” for “ $\neg \Box_r \neg \varphi_1$ ”, “ $\varphi_1 \vee \varphi_2$ ” for “ $\neg(\neg \varphi_1 \wedge \neg \varphi_2)$ ”, “ $\varphi_1 \rightarrow \varphi_2$ ” for “ $\neg(\varphi_1 \wedge \neg \varphi_2)$ ”, “ $\varphi_1 \leftrightarrow \varphi_2$ ” for “ $\neg(\varphi_1 \wedge \neg \varphi_2) \wedge \neg(\varphi_2 \wedge \neg \varphi_1)$ ”, “ \top ” and \perp for the true and false constants respectively. Formulas like $\neg\neg\psi$ are implicitly assumed to be simplified into ψ ; thus, if ψ is $\neg\phi$, then by “ $\neg\psi$ ” we mean “ ϕ ”. We often write “ $(\bigwedge_i l_i) \rightarrow \bigvee_j l_j$ ” for the clause “ $\bigvee_j \neg l_i \vee \bigvee_j l_j$ ”, and “ $(\bigwedge_i l_i) \rightarrow (\bigwedge_j l_j)$ ” for the conjunction of clauses “ $\bigwedge_j (\bigvee_i \neg l_i \vee l_j)$ ”. We call *depth* of φ , written $depth(\varphi)$, the maximum degree of nesting of modal operators in φ .

A K_m -formula is said to be in *Negative Normal Form (NNF)* if it is written in terms of the symbols $\Box_r, \diamond_r, \wedge, \vee$ and propositional literals $A_i, \neg A_i$ (i.e., if all negations occur only before propositional atoms in \mathcal{A}). Every K_m -formula φ can be converted into an equivalent one $NNF(\varphi)$ by recursively applying the rewriting rules: $\neg \Box_r \varphi \implies \diamond_r \neg \varphi$, $\neg \diamond_r \varphi \implies \Box_r \neg \varphi$, $\neg(\varphi_1 \wedge \varphi_2) \implies (\neg \varphi_1 \vee \neg \varphi_2)$, $\neg(\varphi_1 \vee \varphi_2) \implies (\neg \varphi_1 \wedge \neg \varphi_2)$, $\neg\neg\varphi \implies \varphi$.

A K_m -formula is said to be in *Box Normal Form (BNF)* [PSV02, PV03] if it is written in terms of the symbols $\Box_r, \neg \Box_r, \wedge, \vee$, and propositional literals $A_i, \neg A_i$ (i.e., if there are no diamonds, and if all negations occurs only before boxes or before propositional atoms in \mathcal{A}). Every K_m -formula φ can be converted into an equivalent one $BNF(\varphi)$ by recursively applying the rewriting rules: $\diamond_r \varphi \implies \neg \Box_r \neg \varphi$, $\neg(\varphi_1 \wedge \varphi_2) \implies (\neg \varphi_1 \vee \neg \varphi_2)$, $\neg(\varphi_1 \vee \varphi_2) \implies (\neg \varphi_1 \wedge \neg \varphi_2)$, $\neg\neg\varphi \implies \varphi$.

The basic normal modal logic K_m can be defined axiomatically as follows. A formula in Λ_m is a theorem in K_m if it can be inferred from the following axiom

schema:

$$K. (\Box_r \varphi_1 \wedge \Box_r(\varphi_1 \rightarrow \varphi_2)) \rightarrow \Box_r \varphi_2. \quad (32.1)$$

by means of tautological inference and of the application of the following inference rule:

$$\frac{\varphi}{\Box_r \varphi} \text{ (Necessitation)}, \quad (32.2)$$

for every formula $\varphi, \varphi_1, \varphi_2$ in Λ_m and for every $\Box_r \in \mathcal{B}$. The rule *Necessitation* characterizes most modal logics; the axiom *K* characterizes the normal modal logics.

The semantics of modal logics is given by means of Kripke structures. A *Kripke structure* for K_m is a tuple $M = \langle \mathcal{U}, \pi, \mathcal{R}_1, \dots, \mathcal{R}_m \rangle$, where \mathcal{U} is a set of states, π is a function $\pi : \mathcal{A} \times \mathcal{U} \mapsto \{\text{True}, \text{False}\}$, and each \mathcal{R}_r is a binary relation on the states of \mathcal{U} . With a little abuse of notation we write “ $u \in M$ ” instead of “ $u \in \mathcal{U}$ ”. We call a pair M, u , a *situation*. The binary relation \models between a modal formula φ and a pair M, u s.t. $u \in M$ is defined as follows:

$$\begin{aligned} M, u \models A_i, A_i \in \mathcal{A} &\iff \pi(A_i, u) = \text{True}; \\ M, u \models \neg \varphi_1 &\iff M, u \not\models \varphi_1; \\ M, u \models \varphi_1 \wedge \varphi_2 &\iff M, u \models \varphi_1 \text{ and } M, u \models \varphi_2; \\ M, u \models \Box_r \varphi_1, \Box_r \in \mathcal{B} &\iff M, v \models \varphi_1 \text{ for every } v \in M \text{ s.t. } \mathcal{R}_r(u, v) \text{ holds in } M. \end{aligned}$$

We extend the definition of \models to formula sets $\mu = \{\varphi_1, \dots, \varphi_n\}$ as follows:

$$M, u \models \mu \iff M, u \models \varphi_i, \text{ for every } \varphi_i \in \mu.$$

“ $M, u \models \varphi$ ” should be read as “ M, u satisfy φ in K_m ” (alternatively, “ M, u K_m -satisfy φ ”). We say that a formula $\varphi \in \Lambda_m$ is satisfiable in K_m (K_m -satisfiable from now on) if and only if there exist M and $u \in M$ s.t. $M, u \models \varphi$. φ is *valid* for M , written $M \models \varphi$, if $M, u \models \varphi$ for every $u \in M$. φ is valid for a class of Kripke structures \mathcal{K} if $M \models \varphi$ for every $M \in \mathcal{K}$. φ is said to be *valid in K_m* iff $M \models \varphi$ for every Kripke structure M . It can be proved that a Λ_m -formula φ is a theorem in K_m if and only if it is valid in K_m [Che80, Fit83, HM92].

When this causes no ambiguity we sometimes write “satisfiability” meaning “ K_m -satisfiability”. If $m = 1$, we simply write “ K ” for “ K_1 ”.

The problem of determining the K_m -satisfiability of a K_m -formula φ is decidable and PSPACE-complete [Lad77, HM92], even restricting the language to a single Boolean atom (i.e., $\mathcal{A} = \{A_1\}$) [Hal95]; if we impose a bound on the modal depth of the K_m -formulas, the problem reduces to NP-complete [Hal95]. Intuitively, every satisfiable formula φ in K_m can be satisfied by a Kripke structure M which is a finite tree and whose depth is given by $\text{depth}(\varphi) + 1$ (i.e., s.t. $|M| \leq |\varphi|^{\text{depth}(\varphi)}$). Such a structure can be spanned by an alternating-and/or search procedure, similarly to what is done with QBF. An encoding of K_m -satisfiability into QBF is presented in [Lad77, HM92]. For a detailed description on K_m , including complexity results, we refer the reader to [Lad77, HM92, Hal95].

Table 32.1. Axiom schemata and corresponding properties of \mathcal{R}_r for the normal modal logics.

Axiom Schema	Property of \mathcal{R}_r
B. $\neg\varphi \rightarrow \Box_r\neg\Box_r\varphi$	symmetric $\forall u v. [\mathcal{R}_r(u, v) \implies \mathcal{R}_r(v, u)]$
D. $\neg\Box_r\perp$	seriality $\forall u. \exists v. [\mathcal{R}_r(u, v)]$
T. $\Box_r\varphi \rightarrow \varphi$	reflexive $\forall u. [\mathcal{R}_r(u, u)]$
4. $\Box_r\varphi \rightarrow \Box_r\Box_r\varphi$	transitive $\forall u v w. [\mathcal{R}_r(u, v) \wedge \mathcal{R}_r(v, w) \implies \mathcal{R}_r(u, w)]$
5. $\neg\Box_r\varphi \rightarrow \Box_r\neg\Box_r\varphi$	euclidean $\forall u v w. [\mathcal{R}_r(u, v) \wedge \mathcal{R}_r(u, w) \implies \mathcal{R}_r(v, w)]$

Table 32.2. Properties of \mathcal{R}_r for the various normal modal logics. The names between parentheses denote the names each logic is commonly referred with. (For better readability, we omit the pedex “ m ” from the name of the logics.)

Logic $\mathcal{L} \in \mathcal{N}$ (Axiomatic Characterization)	Corresponding Properties of \mathcal{R}_r (Semantic Characterization)
K	—
KB	symmetric
KD	serial
KT = KDT (T)	reflexive
K4	transitive
K5	euclidean
KBD	symmetric and serial
KBT = KBDT (B)	symmetric and reflexive
KB4 = KB5 = KB45	symmetric and transitive
KD4	serial and transitive
KD5	serial and euclidean
KT4 = KDT4 (S4)	reflexive and transitive
KT5 = KBD4 = KBD5 = KBT4 = KBT5 = KDT5 = KT45 = KBD45 = KBT45 = KDT45 = KBDT4 = KBDT5 = KBDT45 (S5)	reflexive, transitive and symmetric (equivalence)
K45	transitive and euclidean
KD45	serial, transitive and euclidean

32.2.2. Normal Modal Logics

We consider the class \mathcal{N} of the normal modal logics. We briefly recall some of the standard definitions and results for these logics (see, e.g., [Che80, Fit83, HM92, Hal95]).

Given the language Λ_m , the class of normal modal logics on Λ_m , \mathcal{N} , can be described axiomatically as follows. The set of theorems in a logic \mathcal{L} in \mathcal{N} is the set of Λ_m -formulas which can be inferred by means of tautological inference and of the application of the Necessitation rule from the axiom schema K , plus a given subset of the axiom schemata $\{B, D, T, 4, 5\}$ described in the left column of Table 32.1. A list of normal modal logics built by combining such axiom schemata is presented in the left column of Table 32.2. Notice that each logic \mathcal{L} is named after the list of its (modal) axiom schemata, and that many logics are equivalent, so that we have only 15 distinct logics out of the 32 possible combinations.

From the semantic point of view, the logics $\mathcal{L} \in \mathcal{N}$ differ from one another by imposing some restrictions on the relations \mathcal{R}_r of the Kripke structures. As described in Table 32.1, each axiom schema in $\{B, D, T, 4, 5\}$ corresponds to a property on \mathcal{R}_r . In each logic \mathcal{L} , a formula φ can be satisfied only by Kripke structures whose relations \mathcal{R}_r 's verify the properties corresponding to \mathcal{L} , as described in Table 32.2. (E.g., φ is satisfiable in $KD4$ only by Kripke structures whose relations are both serial and transitive.) Consequently, φ is said to be *valid in \mathcal{L}* if it is valid in the corresponding class of Kripke structures. For every \mathcal{L} in \mathcal{N} , it can be proved that a Λ_m -formula is a theorem in \mathcal{L} if and only if it is valid in \mathcal{L} [Che80, Fit83].

The problem of determining the satisfiability in \mathcal{L} in \mathcal{N} (“ \mathcal{L} -satisfiability” hereafter) of a Λ_m -formula φ is decidable for every \mathcal{L} . The computational complexity of the problem depends on the logic \mathcal{L} and on many other factors, including the maximum number m of distinct box operators, the maximum number $|\mathcal{A}|$ of distinct primitive propositions, and the maximum modal depth of the formulas (denoted by *depth*). In the general case, for most logics $\mathcal{L} \in \mathcal{N}$ \mathcal{L} -satisfiability is PSPACE-complete; in some cases it may reduce to NP-complete, if $m = 1$ (e.g., with $K45$, $KD45$, $S5$), if *depth* is bounded (e.g., with K_m , T_m , $K45_m$, $KD45_m$, $S5_m$); in some cases it may reduce even to PTIME-complete is some of the features above combine with the fact that $|\mathcal{A}|$ is finite [Lad77, Hal95]. We refer the reader to [Lad77, HM92, Hal95, Ngu05] for a detailed description of these issues.

A *labeled formula* is a pair $\sigma : \varphi$, where φ is a formula in Λ and σ is a *label* (typically a sequence of integers) labeling a world in a Kripke structure for \mathcal{L} . If $\Gamma = \{\varphi_1, \dots, \varphi_n\}$, we write $\sigma : \Gamma$ for $\{\sigma : \varphi_1, \dots, \sigma : \varphi_n\}$. Intuitively, $\sigma : \varphi$ means “the formula φ in the world σ ”. For every $\mathcal{L} \in \mathcal{N}$, [Fit83, Mas94, Mas00] give a notion of *accessibility relation* between labels and gives the properties for these relations for the various logics \mathcal{L} . Essentially, they mirror the accessibility relation between the worlds they label.

32.2.3. Non-normal Modal Logics

We now consider the class of classical – also known as non-normal – modal logics. We briefly recall some of the standard definitions and results for these logics (see, e.g., [Che80, FHMV95]).

Given the language Λ_m , the basic classical modal logic on Λ_m , E_m , can be defined axiomatically as follows. The theorems in E_m are the set of formulas in Λ_m which can be inferred by tautological inference and by the application of the inference rule:

$$\frac{\varphi \leftrightarrow \psi}{\Box_r \varphi \leftrightarrow \Box_r \psi} \quad (E). \quad (32.3)$$

As a consequence, the schemata

$$\begin{aligned} N. & \Box_r \top \\ M. & \Box_r(\varphi \wedge \psi) \rightarrow \Box_r \varphi \\ C. & (\Box_r \varphi \wedge \Box_r \psi) \rightarrow \Box_r(\varphi \wedge \psi) \end{aligned} \quad (32.4)$$

which are theorems in K_m do not hold in E_m . The three principles N , M , and C enforce closure conditions on the set of provable formulas which are not always desirable, especially if the \Box_r operator has an epistemic (such as knowledge or belief)

reading. If we interpret $\Box_r\varphi$ as “a certain agent r believes φ ”, then N enforces that r believes all the logical truths, M that r 's beliefs are closed under logical consequence, and C that r 's beliefs are closed under conjunction. These three closure properties are different forms of omniscience, and —as such— they might not be appropriate for modeling the beliefs of a real agent (see, e.g., [FHMV95]). By combining the schemata in (32.4) and using them as axiom schemata, we can get eight different combinations corresponding to eight distinct logics, where each logic is named after the list of its modal axiom schemata. The logic $EMCN_m$ corresponds to the basic normal modal logic K_m .

The semantics of classical modal logics is given by means of Montague-Scott structures. A *Montague-Scott structure* for E_m is a tuple $S = \langle \mathcal{U}, \pi, \mathcal{N}_1, \dots, \mathcal{N}_m \rangle$, where \mathcal{U} is a set of states, π is a function $\pi : \mathcal{A} \times \mathcal{U} \mapsto \{True, False\}$, and each \mathcal{N}_r is a relation $\mathcal{N}_r : \mathcal{U} \mapsto \mathcal{P}(\mathcal{P}(\mathcal{U}))$, i.e., for each $u \in \mathcal{U}$, $\mathcal{N}_r(u) \subseteq \mathcal{P}(\mathcal{U})$. Notice that Montague-Scott structures are a generalization of Kripke structures, so the class of possible models for K_m is indeed a subclass of the possible models for E_m . In analogy with Section 32.2.1, we write “ $u \in S$ ” instead of “ $u \in \mathcal{U}$ ”, and we call S, u a situation. The binary relation \models between a modal formula φ and a pair S, u s.t. $u \in S$ is defined as follows:

$$\begin{aligned}
 S, u \models A_i, A_i \in \mathcal{A} &\iff \pi(A_i, u) = True; \\
 S, u \models \neg\varphi_1 &\iff S, u \not\models \varphi_1; \\
 S, u \models \varphi_1 \wedge \varphi_2 &\iff S, u \models \varphi_1 \text{ and } S, u \models \varphi_2; \\
 S, u \models \Box_r\varphi_1, \Box_r \in \mathcal{B} &\iff \{v \mid M, v \models \varphi_1\} \in \mathcal{N}_r(u)
 \end{aligned}
 \tag{32.5}$$

We extend the definition of \models to formula sets $\mu = \{\varphi_1, \dots, \varphi_n\}$ as follows:

$$S, u \models \mu \iff S, u \models \varphi_i, \text{ for every } \varphi_i \in \mu.$$

“ $S, u \models \varphi$ ” should be read as “ S, u satisfy φ in E_m ” (alternatively, “ S, u E_m -satisfy φ ”). We say that a formula $\varphi \in \Lambda_m$ is satisfiable in E_m (E_m -satisfiable from now on) if and only if there exist S and $u \in S$ s.t. $S, u \models \varphi$. φ is *valid* for S , written $S \models \varphi$, if $S, u \models \varphi$ for every $u \in S$. φ is valid for a class of Montague-Scott structures \mathcal{C} if $S \models \varphi$ for every $S \in \mathcal{C}$. φ is said to be *valid in E_m* iff $S \models \varphi$ for every Montague-Scott structure S . The semantics of the logic E is given by the relation defined in 32.5 only. The logics where one of M , C and N is an axiom require the following closure conditions on \mathcal{N}_r to be satisfied for each r :

- (M) if $U \subseteq V$ and $U \in \mathcal{N}_r(w)$ then $V \in \mathcal{N}_r(w)$ (closure by superset inclusion),
- (C) if $U \in \mathcal{N}_r(w)$ and $V \in \mathcal{N}_r(w)$ then $U \cap V \in \mathcal{N}_r(w)$ (closure by intersection),
- (N) $\mathcal{U} \in \mathcal{N}_r(w)$ (unit containment).

Notice that if an E_m structure S is such that $\mathcal{N}_r(u)$ satisfies all the above conditions for each world $u \in S$, then S is also a K_m structure.

In analogy with Section 32.2.1, if $m = 1$, we simply write, e.g., “ E ” for “ E_1 ”, “ EM ” for “ EM_1 ”, and so on. For every non-normal logic \mathcal{L} , it can be proved that a Λ_m -formula is a theorem in \mathcal{L} if and only if it is valid in \mathcal{L} [Che80, Fit83].

The problem of determining the E_m -satisfiability of a E_m -formula φ is decidable, and the E_m -satisfiability problem is NP-complete [Var89]. Satisfiability

is also NP-complete in all the classical modal logics that do not contain C as an axiom (EM, EN, EMN), while it is PSPACE-complete in the remaining ones (EC, EMC, ECN, EMCN). The satisfiability problems maintain the same complexity classes when considering multi-agent extensions.

32.2.4. Modal Logics and Description Logics

The connection between modal logics and terminological logics – also known as description logics – is due to a seminal paper by Klaus Schild [Sch91] where the description logic \mathcal{ALC} [SSS91] is shown to be a notational variant of the modal logic K_m . Here we survey some of the results of [Sch91], and we refer the reader to [BCM⁺03] for further reading about the current state of the art in modal and description logics.

Following [Sch91], we start by defining the language of \mathcal{ALC} . Formally, the language \mathcal{ALC} is defined by grammar rules of the form:

$$\begin{aligned} C &\rightarrow c \mid \top \mid C_1 \sqcap C_2 \mid \neg C \mid \forall R.C \\ R &\rightarrow r \end{aligned} \tag{32.6}$$

where C , and C_i denote generical concepts, c denotes an atomic concept symbol and r a role symbol. The formal semantics of \mathcal{ALC} is specified by an *extension function*. Let \mathcal{D} be any set called the *domain*. An extension function ε over \mathcal{D} is a function mapping concepts to subsets of \mathcal{D} and roles to subsets of $\mathcal{D} \times \mathcal{D}$ such that

$$\begin{aligned} \varepsilon[\top] &= \mathcal{D} \\ \varepsilon[C \sqcap D] &= \varepsilon[C] \cap \varepsilon[D] \\ \varepsilon[\neg C] &= \mathcal{D} \setminus \varepsilon[C] \\ \varepsilon[\forall R.C] &= \{d \in \mathcal{D} \mid \forall \langle d, e \rangle \in \varepsilon[R] \quad e \in \varepsilon[C]\} \end{aligned} \tag{32.7}$$

Using extension functions, we can define the semantic notion of *subsumption*, *equivalence* and *coherence*: D *subsumes* C , written $\models C \sqsubseteq D$, iff for each extension function ε , $\varepsilon[C] \subseteq \varepsilon[D]$, whereas C and D are *equivalent*, written $\models C = D$ iff for each extension function ε , $\varepsilon[C] = \varepsilon[D]$. Finally, C is *coherent* iff there is an extension function ε with $\varepsilon[C] \neq \emptyset$. The following result allows us to concentrate on any of the above notions without loss of generality:

Lemma 1. [Sch91] *Subsumption, equivalence, and incoherence are log-space reducible to each other in any terminological logic comprising Boolean operations on concepts.*

Viewing \mathcal{ALC} from the modal logic perspective (see 32.2.1), atomic concepts simply can be expounded as atomic propositions, and can be interpreted as the set of states in which such propositions hold. In this case “ \forall .” becomes a modal operator since it is applied to formulas. Thus, e.g., $\neg c_1 \sqcup \forall r.(c_2 \sqcap c_3)$ can be expressed by the K_m -formula $\neg A_1 \vee \square_r(A_2 \wedge A_3)$. The subformula $\square_r(A_2 \wedge A_3)$ is to be read as “agent r knows $A_2 \wedge A_3$ ”, and means that in every state accessible for r , both A_2 and A_3 hold.² Actually

²Notice that we replaced primitive concepts c_i with $i \in \{1, 2, 3\}$ with propositions A_i , assuming the obvious bijection between the two sets.

- the domain of an extension function can be read as a set of states \mathcal{U} ,
- atomic concepts can be interpreted as the set of worlds in which they hold, if expounded as atomic formulas, and
- atomic roles can be interpreted as accessibility relations.

Hence $\forall R.C$ can be expounded as “all states in which agent R knows proposition C ” instead of “all objects for which all R 's are in C ”.

To establish the correspondence between \mathcal{ALC} and K_m consider the function f mapping \mathcal{ALC} concepts to K_m -formulas with $f(c_i) = A_i$ for $i \in 1, 2, \dots$, i.e., f maps concept symbols to primitive propositions, $f(\top) = \top$, $f(C \sqcap D) = f(C) \wedge f(D)$, $f(\neg C) = \neg f(C)$ and $f(\forall R.C) = \Box_R f(C)$. It could easily be shown by induction on the complexity of \mathcal{ALC} -concepts that f is a linearly length-bounded isomorphism such that an \mathcal{ALC} -concept C is coherent iff the K_m -formula $f(C)$ is satisfiable. Formally:

Theorem 1. [Sch91] *\mathcal{ALC} is a notational variant of the propositional modal logic K_m , and satisfiability in K_m has the same computational complexity as coherence in \mathcal{ALC} .*

By this correspondence, several theoretical results for K_m can easily be carried over to \mathcal{ALC} . We immediately know, for example, that without loss of generality, any decision procedure for K_m -satisfiability is also a decision procedure for \mathcal{ALC} -coherence.

There are other result (see, e.g., [BCM⁺03]) that link normal modal logics, as described in 32.2.2, to various description logics that extend \mathcal{ALC} in several ways. According to these results, decision procedures for expressive description logics may be regarded as decision procedures for various normal modal logics, e.g., KD, T, B, and the other way round.

32.3. Basic Modal DPLL

In this section we introduce the basic concepts of modal tableau-based and DPLL-based procedures, and we discuss their relation.

32.3.1. A Formal Framework

Assume w.l.o.g. that the \diamond_r 's are not part of the language (each $\diamond_r \varphi$ can be rewritten into $\neg \Box_r \neg \varphi$). We call *atom* every formula that cannot be decomposed propositionally, that is, every formula whose main connective is not propositional. Examples of atoms are, A_1, A_i (*propositional atoms*), $\Box_1(A_1 \vee \neg A_2)$ and $\Box_2(\Box_1 A_1 \vee \neg A_2)$ (*modal atoms*). A *literal* is either an atom or its negation. Given a formula φ , an atom [literal] is a *top-level atom [literal]* for φ if and only if it occurs in φ and under the scope of no boxes. $Atoms^0(\varphi)$ is the set of the top-level atoms of φ .

We call a *truth assignment* μ for a formula φ a truth value assignment to all the atoms of φ . A truth assignment is *total* if it assigns a value to all atoms in φ , *partial* otherwise. Syntactically identical instances of the same atom are always assigned identical truth values; syntactically different atoms, e.g., $\Box_1(\varphi_1 \vee \varphi_2)$

and $\Box_1(\varphi_2 \vee \varphi_1)$, are treated differently and may thus be assigned different truth values.

To this extent, we introduce a bijective function $\mathcal{L2P}$ (“ \mathcal{L} -to-Propositional”) and its inverse $\mathcal{P2L} := \mathcal{L2P}^{-1}$ (“Propositional-to- \mathcal{L} ”), s.t. $\mathcal{L2P}$ maps top-level Boolean atoms into themselves and top-level non-Boolean atoms into fresh Boolean atoms — so that two atom instances in φ are mapped into the same Boolean atom iff they are syntactically identical— and distributes with sets and Boolean connectives. (E.g., $\mathcal{L2P}(\{\Box_r\varphi_1, \neg(\Box_r\varphi_1 \vee \neg A_1)\})$ is $\{B_1, \neg(B_1 \vee \neg A_1)\}$.) $\mathcal{L2P}$ and $\mathcal{P2L}$ are also called *Boolean abstraction* and *Boolean refinement* respectively.

We represent a truth assignment μ for φ as a set of literals

$$\begin{aligned} \mu = \{ & \Box_1\alpha_{11}, \dots, \Box_1\alpha_{1N_1}, \neg\Box_1\beta_{11}, \dots, \neg\Box_1\beta_{1M_1}, \\ & \vdots \\ & \Box_m\alpha_{m1}, \dots, \Box_m\alpha_{mN_m}, \neg\Box_m\beta_{m1}, \dots, \neg\Box_m\beta_{mM_m}, \\ & A_1, \dots, \neg A_R, \neg A_{R+1}, \dots, \neg A_S\}, \end{aligned} \quad (32.8)$$

$\Box_r\alpha_i$'s, $\Box_r\beta_j$'s being modal atoms and A_i 's being propositional atoms. Positive literals $\Box_r\alpha_i$ and A_k in μ mean that the corresponding atom is assigned to true, negative literals $\neg\Box_r\beta_i$ and $\neg A_k$ mean that the corresponding atom is assigned to false. If $\mu_2 \subseteq \mu_1$, then we say that μ_1 *extends* μ_2 and that μ_2 *subsumes* μ_1 . A *restricted truth assignment*

$$\mu^r = \{\Box_r\alpha_{r1}, \dots, \Box_r\alpha_{rN_r}, \neg\Box_r\beta_{r1}, \dots, \neg\Box_r\beta_{rM_r}\} \quad (32.9)$$

is given by restricting μ to the set of atoms in the form $\Box_r\psi$, where $1 \leq r \leq m$. Trivially μ^r subsumes μ .

Notationally, we use the Greek letters μ, η to represent truth assignments. Sometimes we represent the truth assignments in (32.8) and (32.9) also as the formulas given by the conjunction of their literals:

$$\mu = \begin{cases} \bigwedge_{i=1}^{N_1} \Box_1\alpha_{1i} \wedge \bigwedge_{j=1}^{M_1} \neg\Box_1\beta_{1j} \wedge \\ \dots \\ \bigwedge_{i=1}^{N_m} \Box_m\alpha_{mi} \wedge \bigwedge_{j=1}^{M_m} \neg\Box_m\beta_{mj} \wedge \\ \bigwedge_{k=1}^R A_k \wedge \bigwedge_{h=R+1}^S \neg A_h, \end{cases} \quad (32.10)$$

$$\mu^r = \bigwedge_i \Box_r\alpha_{ri} \wedge \bigwedge_j \neg\Box_r\beta_{rj}. \quad (32.11)$$

For every logic \mathcal{L} , we say that an assignment μ [restricted assignment μ^r] is \mathcal{L} -satisfiable meaning that its corresponding formula (32.10) [(32.11)] is \mathcal{L} -satisfiable.

We say that a total truth assignment μ for φ *propositionally satisfies* φ , written $\mu \models_p \varphi$, if and only if $\mathcal{L2P}(\mu) \models \mathcal{L2P}(\varphi)$, that is, for all sub-formulas

φ_1, φ_2 of φ :

$$\begin{aligned} \mu \models_p \varphi_1, \varphi_1 \in \text{Atoms}^0(\varphi) &\iff \varphi_1 \in \mu, \\ \mu \models_p \neg\varphi_1 &\iff \mu \not\models_p \varphi_1, \\ \mu \models_p \varphi_1 \wedge \varphi_2 &\iff \mu \models_p \varphi_1 \text{ and } \mu \models_p \varphi_2. \end{aligned}$$

We say that a partial truth assignment μ *propositionally satisfies* φ if and only if all the total truth assignments for φ which extend μ propositionally satisfy φ . For instance, if $\varphi = \Box_1\varphi_1 \vee \neg\Box_2\varphi_2$, then the partial assignment $\mu = \{\Box_1\varphi_1\}$ is such that $\mu \models_p \varphi$. In fact, both $\{\Box_1\varphi_1 \Box_2\varphi_2\}$ and $\{\Box_1\varphi_1, \neg\Box_2\varphi_2\}$ propositionally satisfy φ . Henceforth, if not otherwise specified, when dealing with propositional satisfiability we do not distinguish between assignments and partial assignments. Intuitively, if we consider a formula φ as a propositional formula in its top-level atoms, then \models_p is the standard satisfiability in propositional logic. Thus, for every φ_1 and φ_2 , we say that $\varphi_1 \models_p \varphi_2$ if and only if $\mu \models_p \varphi_2$ for every μ s.t. $\mu \models_p \varphi_1$. We say that φ is *propositionally satisfiable* if and only if there exist an assignment μ s.t. $\mu \models_p \varphi$. We also say that $\models_p \varphi$ (φ is *propositionally valid*) if and only if $\mu \models_p \varphi$ for every assignment μ for φ . Thus $\varphi_1 \models_p \varphi_2$ if and only if $\models_p \varphi_1 \rightarrow \varphi_2$, and $\models_p \varphi$ iff $\neg\varphi$ is propositionally unsatisfiable. Notice that \models_p is stronger than \models , that is, if $\varphi_1 \models_p \varphi_2$, then $\varphi_1 \models \varphi_2$, but not vice versa. E.g., $\Box_r\varphi_1 \wedge \Box_r(\varphi_1 \rightarrow \varphi_2) \models \Box_r\varphi_2$, but $\Box_r\varphi_1 \wedge \Box_r(\varphi_1 \rightarrow \varphi_2) \not\models_p \Box_r\varphi_2$.

Example 1. Consider the following K_2 formula φ and its Boolean abstraction $\mathcal{L}2\mathcal{P}(\varphi)$:

$$\begin{aligned} \varphi = & \{ \neg\Box_1(\neg A_3 \vee \neg A_1 \vee A_2) \vee A_1 \vee A_5 \} \\ & \wedge \{ \neg A_2 \vee \neg A_5 \vee \Box_1(\neg A_2 \vee A_4 \vee A_5) \} \\ & \wedge \{ A_1 \vee \Box_2(\neg A_4 \vee A_5 \vee A_2) \vee A_2 \} \\ & \wedge \{ \neg\Box_2(A_4 \vee \neg A_3 \vee A_1) \vee \neg\Box_1(A_4 \vee \neg A_2 \vee A_3) \vee \neg A_5 \} \\ & \wedge \{ \neg A_3 \vee A_1 \vee \Box_2(\neg A_4 \vee A_5 \vee A_2) \} \\ & \wedge \{ \Box_1(\neg A_5 \vee A_4 \vee A_3) \vee \Box_1(\neg A_1 \vee A_4 \vee A_3) \vee \neg A_1 \} \\ & \wedge \{ A_1 \vee \Box_1(\neg A_2 \vee A_1 \vee A_4) \vee A_2 \} \end{aligned}$$

$$\begin{aligned} \mathcal{L}2\mathcal{P}(\varphi) = & \{ \neg B_1 \vee A_1 \vee A_5 \} \\ & \wedge \{ \neg A_2 \vee \neg A_5 \vee B_2 \} \\ & \wedge \{ A_1 \vee B_3 \vee A_2 \} \\ & \wedge \{ \neg B_4 \vee \neg B_5 \vee \neg A_5 \} \\ & \wedge \{ \neg A_3 \vee A_1 \vee B_3 \} \\ & \wedge \{ B_6 \vee B_7 \vee \neg A_1 \} \\ & \wedge \{ A_1 \vee B_8 \vee A_2 \} \end{aligned}$$

The partial assignment $\mu^p = \{B_6, B_8, \neg B_1, \neg B_5, B_3, \neg A_2\}$ satisfies $\mathcal{L}2\mathcal{P}(\varphi)$, so that the following assignment $\mu := \mathcal{P}2\mathcal{L}(\mu^p)$ propositionally satisfies φ :

$$\begin{aligned} \mu = & \Box_1(\neg A_5 \vee A_4 \vee A_3) \wedge \Box_1(\neg A_2 \vee A_1 \vee A_4) \wedge \left[\bigwedge_i \Box_1 \alpha_{1i} \right] \\ & \neg\Box_1(\neg A_3 \vee \neg A_1 \vee A_2) \wedge \neg\Box_1(A_4 \vee \neg A_2 \vee A_3) \wedge \left[\bigwedge_j \neg\Box_1 \beta_{1j} \right] \\ & \Box_2(\neg A_4 \vee A_5 \vee A_2) \wedge \left[\bigwedge_i \Box_2 \alpha_{2i} \right] \\ & \neg A_2. \left[\bigwedge_k A_k \wedge \bigwedge_h \neg A_h \right] \end{aligned}$$

μ gives rise to two restricted assignments μ^1 and μ^2 :

$$\begin{aligned} \mu^1 &= \Box_1(\neg A_5 \vee A_4 \vee A_3) \wedge \Box_1(\neg A_2 \vee A_1 \vee A_4) \wedge [\bigwedge_i \Box_1 \alpha_{1i}] \\ &\quad \neg \Box_1(\neg A_3 \vee \neg A_1 \vee A_2) \wedge \neg \Box_1(A_4 \vee \neg A_2 \vee A_3) \quad [\bigwedge_j \neg \Box_1 \beta_{1j}] \\ \mu^2 &= \Box_2(\neg A_4 \vee A_5 \vee A_2) \quad [\bigwedge_i \Box_2 \alpha_{2i}]. \end{aligned}$$

We say that a collection $\mathcal{M} := \{\mu_1, \dots, \mu_n\}$ of (possibly partial) assignments propositionally satisfying φ is *complete* if and only if, for every total assignment η s.t. $\eta \models_p \varphi$, there exists $\mu_j \in \mathcal{M}$ s.t. $\mu_j \subseteq \eta$. Intuitively, \mathcal{M} can be seen as a compact representation of the whole set of total assignments propositionally satisfying φ .

Proposition 1. [SV98] *Let φ be a formula and let $\mathcal{M} := \{\mu_1, \dots, \mu_n\}$ be a complete collection of truth assignments propositionally satisfying φ . Then, for every \mathcal{L} , φ is \mathcal{L} -satisfiable if and only if μ_j is \mathcal{L} -satisfiable for some $\mu_j \in \mathcal{M}$.*

We also notice the following fact.

Proposition 2. [Seb01] *Let α be a non-Boolean atom occurring only positively [resp. negatively] in φ . Let \mathcal{M} be a complete set of assignments satisfying φ , and let*

$$\mathcal{M}' := \{\mu_j \setminus \{\neg\alpha\} \mid \mu_j \in \mathcal{M}\} \quad [\text{resp. } \{\mu_j \setminus \{\alpha\} \mid \mu_j \in \mathcal{M}\}].$$

Then (i) for every $\mu'_j \in \mathcal{M}'$, $\mu'_j \models_p \varphi$, and (ii) φ is \mathcal{L} -satisfiable if and only if there exist a \mathcal{L} -satisfiable $\mu'_j \in \mathcal{M}'$.

proposition 1 shows that the \mathcal{L} -satisfiability of a formula can be reduced to that of a complete collection of sets of literals (assignment), for every call. Proposition 2 says that, if we have non-Boolean atoms occurring only positively [resp. negatively] in the input formula, we can safely drop every negative [resp. positive] occurrence of them from all assignments in a complete set \mathcal{M} preserving the completeness of \mathcal{M} . In general, \mathcal{L} -satisfiability of a conjunction of literals depends on \mathcal{L} [Fit83, Che80]. The following propositions give a recursive definition for K_m .

Proposition 3. [GS00] *The truth assignment μ of Equation (32.10) is K_m -satisfiable if and only if the restricted truth assignment μ^r of Equation (32.11) is K_m -satisfiable, for all \Box_r 's.³*

Proposition 4. [GS00] *The restricted assignment μ^r of Equation (32.11) is K_m -satisfiable if and only if the formula*

$$\varphi^{rj} = \bigwedge_i \alpha_{ri} \wedge \neg \beta_{rj} \tag{32.12}$$

is K_m -satisfiable, for every $\neg \Box_r \beta_{rj}$ occurring in μ^r .

Notice that propositions 3 and 4 can be merged into one single theorem stating that μ is K_m -satisfiable if and only if φ^{rj} is K_m -satisfiable, for all r and j . Notice furthermore that the depth of every φ^{rj} is strictly smaller than the depth of φ .

³Notice that the component $\bigwedge_{k=1}^R A_k \wedge \bigwedge_{h=R+1}^S \neg A_h$ in (32.10) is consistent because μ is a truth assignment.

Example 2. Consider the formula φ and the assignments μ , μ^1 and μ^2 in Example 1. μ propositionally satisfies φ . Thus, for proposition 1, φ is K_m -satisfiable if μ is K_m -satisfiable. By proposition 3, μ is K_m -satisfiable if and only if both μ^1 and μ^2 are K_m -satisfiable; by proposition 4, μ^2 is trivially K_m -satisfiable, as it contains no negated boxes, and μ^1 is K_m -satisfiable if and only if each of the formulas

$$\begin{aligned}\varphi^{11} &= \bigwedge_i \alpha_{1i} \wedge \neg\beta_{11} = (\neg A_5 \vee A_4 \vee A_3) \wedge (\neg A_2 \vee A_1 \vee A_4) \wedge A_3 \wedge A_1 \wedge \neg A_2, \\ \varphi^{12} &= \bigwedge_i \alpha_{1i} \wedge \neg\beta_{12} = (\neg A_5 \vee A_4 \vee A_3) \wedge (\neg A_2 \vee A_1 \vee A_4) \wedge \neg A_4 \wedge A_2 \wedge \neg A_3\end{aligned}$$

is K_m -satisfiable. As they both are satisfiable propositional formulas, then φ is K_m -satisfiable.

Proposition 1 reduces the \mathcal{L} -satisfiability of a formula φ to the \mathcal{L} -satisfiability of a complete collection of its truth assignments, for every \mathcal{L} . If \mathcal{L} is K_m , propositions 3 and 4 show how to reduce the latter to the K_m -satisfiability of formulas of smaller depth. This process can be applied recursively, decreasing the depth of the formula considered at each iteration. Following these observations, it is possible to test the K_m -satisfiability of a formula φ by implementing a recursive alternation of two basic steps [GS96a, GS96b]:

1. Propositional reasoning: using some procedure for propositional satisfiability, find a truth assignment μ for φ s.t. $\mu \models_p \varphi$;
2. Modal reasoning: check the K_m -satisfiability of μ by generating the corresponding restricted assignments μ_r 's and formulas φ^{rj} 's.

The two steps recurse down until we get to a truth assignment with no modal atoms. At each level, the process is repeated until either a K_m -satisfiable assignment is found (in which case φ is K_m -satisfiable) or no more assignments are found (in which case φ is not K_m -satisfiable).

32.3.2. Modal Tableaux

We call “tableau-based” a system that implements and extends to other logics the Smullyan’s propositional tableau calculus, as defined in [Smu68]. Tableau-based procedures basically consist of a control strategy applied on top of a tableau framework. By tableau framework for modal logics we denote a refutation formal system extending Smullyan’s propositional tableau with rules handling the modal operators (modal rules). Thus, for instance, in our terminology KRIS [BH91, BFH⁺94], CRACK [BFT95] and LWB [HJSS96] are tableau-based systems.

For instance, in the labeled tableau framework for normal modal logics in \mathcal{N} described in [Fit83, Mas94, Mas00], branches are represented as sets of labeled formulas $u : \psi$, where u labels the state in which the formula ψ has to be satisfiable. At the first step the root $1 : \varphi$ is created, φ being the modal formula to be proved (un)satisfiable. At the i -th step, a branch is expanded by applying to a chosen labeled formula the rule corresponding to its main connective, and adding the resulting labeled formula to the branch. The rules are the following: ⁴

⁴Analogous rules handling negated \wedge 's and \vee 's, double negations $\neg\neg$, single and double implications \rightarrow and \leftrightarrow , diamonds \diamond_r , n-ary \wedge 's and \vee 's, and the negation of all them, can

$$\frac{u : (\varphi_1 \wedge \varphi_2)}{u : \varphi_1, u : \varphi_2} (\wedge) \quad \frac{u : (\varphi_1 \vee \varphi_2)}{u : \varphi_1 \quad u : \varphi_2} (\vee), \quad (32.13)$$

$$\frac{u : \neg \Box_r \varphi}{u' : \neg \varphi} (\neg \Box_r) \quad \frac{u : \Box_r \varphi}{u'' : \varphi} (\Box_r). \quad (32.14)$$

The modal rules are constrained by the following applicability conditions:

- $\neg \Box_r$ -rule: u' is a new state (u' is said to be *directly accessible* from u);
- \Box_r -rule: u'' is an existing state which is accessible from u via \mathcal{R}_r .

Distinct logics \mathcal{L} differ for different notions of accessibility in the \Box_r -rule [Fit83, Mas94, Mas00].

Every application of the \vee -rule splits the branch into two sub-branches. A branch is *closed* when a formula ψ and its negation $\neg \psi$ occur in it. The procedure stops when all branches are closed (φ is \mathcal{L} -unsatisfiable) or no more rule is applicable (φ is \mathcal{L} -satisfiable).

For some modal logics it is possible to drop labels by using alternative sets of non-labeled modal rules [Fit83]. For instance in K_m it is possible to use unlabeled formulas and update branches according to the rules

$$\frac{\Gamma, \varphi_1 \wedge \varphi_2}{\Gamma, \varphi_1, \varphi_2} (\wedge) \quad \frac{\Gamma, \varphi_1 \vee \varphi_2}{\Gamma, \varphi_1 \quad \Gamma, \varphi_2} (\vee) \quad (32.15)$$

$$\frac{\mu}{\alpha_1 \wedge \dots \wedge \alpha_m \wedge \neg \beta_j} (\Box_r / \neg \Box_r) \quad (32.16)$$

for each box-index $r \in \{1, \dots, m\}$. Γ is an arbitrary set of formulas, and μ is a set of literals which includes $\neg \Box_r \beta_j$ and whose only positive \Box_r -atoms are $\Box_r \alpha_1, \dots, \Box_r \alpha_m$.

This describes the tableau-based decision procedure of Figure 32.1, which is the restriction to K_m of the basic version of the KRIS procedure described in [BH91]. Tableau-based formalisms for many modal logics are described, e.g., in [Fit83, Mas94]. Tableau-based procedures for many modal logics are described, e.g., in [BH91, BFH⁺94, BFT95, HJSS96].

32.3.3. From Modal Tableaux to Modal DPLL

We call “DPLL-based” any system that implements and extends to other logics the Davis-Putnam-Logemann-Loveland procedure (DPLL) [DP60, DLL62]. DPLL-based procedures basically consist on the combination of a procedure handling purely-propositional component of reasoning, typically a variant of the DPLL algorithm, and some procedure handling the purely-modal component, typically consisting of a control strategy applied on top of a modal tableau rules.

be derived straightforwardly, and are thus omitted here. Following [Fit83], the \wedge -, \vee -, $\neg \Box_r$ - and \Box_r -rules (and those for their equivalent operators) are often called α -, β -, π -, and ν -rules respectively.

```

function  $K_m$ -Tableau( $\Gamma$ )
  if  $\psi_i \in \Gamma$  and  $\neg\psi_i \in \Gamma$  /* branch closed */
    then return False;
  if  $(\varphi_1 \wedge \varphi_2) \in \Gamma$  /*  $\wedge$ -elimination */
    then return  $K_m$ -Tableau( $\Gamma \cup \{\varphi_1, \varphi_2\} \setminus \{(\varphi_1 \wedge \varphi_2)\}$ );
  if  $(\varphi_1 \vee \varphi_2) \in \Gamma$  /*  $\vee$ -elimination */
    then return
       $K_m$ -Tableau( $\Gamma \cup \{\varphi_1\} \setminus \{(\varphi_1 \vee \varphi_2)\}$ ) or
       $K_m$ -Tableau( $\Gamma \cup \{\varphi_2\} \setminus \{(\varphi_1 \vee \varphi_2)\}$ );
  for every  $r \in \{1, \dots, m\}$  do
    for every  $\neg\Box_r\beta_j \in \Gamma$  do /* branch expanded */
      if not  $K_m$ -Tableau( $\{\neg\beta_j\} \cup \bigcup_{\Box_r\alpha_i \in \Gamma} \{\alpha_i\}$ )
        then return False;
  return True;

```

Figure 32.1. An example of a tableau-based procedure for K_m . We omit the steps for the other operators.

Thus, for instance, in our terminology K_{SAT} [GS96a, GS00], FACT [Hor98b, Hor98a], DLP [PS98], RACER [HM01] are DPLL-based systems.⁵

From a purely-logical viewpoint, it is possible to conceive a DPLL-based framework by substituting the propositional tableaux rules with some rules implementing the DPLL algorithms in a tableau-based framework [SV98]. For instance, one can conceive a DPLL-based framework for a normal logic \mathcal{L} from Fitting or Massacci's frameworks (see §32.3.2) by substituting the \vee -rule (32.13) with the following rules:

$$\frac{u : (l \vee C)}{u : l \quad u : \neg l} \text{ (Branch)} \quad \frac{u : l \quad u : (\neg l \vee C)}{u : C} \text{ (Unit)}, \quad (32.17)$$

where l is a and C is a disjunction of literals.⁶ More recent and richer formal frameworks for representing DPLL and DPLL-based procedures are described in [Tin02, NOT06].

As stated in §32.3.2, for some modal logics it is possible to drop labels by using alternative sets of non-labeled modal rules [Fit83]. If so, DPLL-based procedures can be implemented more straightforwardly. For instance, in K_m it is possible to use unlabeled formulas and update branches according to the following rules [SV98]:

$$\frac{\varphi}{\mu_1 \quad \mu_2 \quad \dots \quad \mu_n} \text{ (DPLL)} \quad \frac{\mu}{\alpha_1 \wedge \dots \wedge \alpha_m \wedge \neg\beta_j} \text{ } (\Box_r / \neg\Box_r) \quad (32.18)$$

where the $\Box_r / \neg\Box_r$ -rule is that of (32.16), and $\{\mu_1, \dots, \mu_n\}$ is a *complete* set of assignments for φ , which can be produced by the DPLL algorithm.

⁵See footnote 1 in §32.1.

⁶Here we assume for simplicity that the input formula is in conjunctive normal form (CNF). Equivalent formalisms are available for non-CNF formulas [DM94, Mas98].

```

function KSAT( $\varphi$ )
  return KSATF( $\varphi$ ,  $\top$ );

function KSATF( $\varphi$ ,  $\mu$ )
  if  $\varphi = \top$                                      /* base */
    then return KSATA( $\mu$ );
  if  $\varphi = \perp$                                      /* backtrack */
    then return False;
  if {a unit clause ( $l$ ) occurs in  $\varphi$ }             /* unit */
    then return KSATF(assign( $l$ ,  $\varphi$ ),  $\mu \wedge l$ );
   $l :=$  choose-literal( $\varphi$ );                         /* split */
  return KSATF(assign( $l$ ,  $\varphi$ ),  $\mu \wedge l$ ) or
        KSATF(assign( $\neg l$ ,  $\varphi$ ),  $\mu \wedge \neg l$ );

/*  $\mu$  is  $\bigwedge_i \Box_i \alpha_i \wedge \bigwedge_j \neg \Box_j \beta_j \wedge \dots \wedge \bigwedge_i \Box_m \alpha_{mi} \wedge \bigwedge_j \neg \Box_m \beta_{mj} \wedge \bigwedge_k A_k \wedge \bigwedge_h \neg A_h$  */
function KSATA( $\mu$ )
  for each box index  $r \in \{1\dots m\}$  do
    if not KSATAR( $\bigwedge_i \Box_r \alpha_{ri} \wedge \bigwedge_j \neg \Box_r \beta_{rj}$ )
      then return False;
  return True;

/*  $\mu^r$  is  $\bigwedge_i \Box_r \alpha_{ri} \wedge \bigwedge_j \neg \Box_r \beta_{rj}$  */
function KSATAR( $\mu^r$ )
  for each literal  $\neg \Box_r \beta_{rj} \in \mu$  do
    if not KSAT( $\bigwedge_i \alpha_{ri} \wedge \neg \beta_{rj}$ )
      then return False;
  return True;

```

Figure 32.2. The basic version of KSAT algorithm.

32.3.4. Basic Modal DPLL for K_m

The ideas described in §32.3.3 were implemented in the KSAT procedure [GS96a, GS00], whose basic version is reported in Figure 32.2. This schema evolved from that of the PTAUT procedure in [AG93], and is based on the “classic” DPLL procedure [DP60, DLL62]. KSAT takes in input a modal formula φ and returns a truth value asserting whether φ is K_m -satisfiable or not. KSAT invokes KSAT_F (where “F” stands for “Formula”), passing as arguments φ and (by reference) the empty assignment \top . KSAT_F tries to build a K_m -satisfiable assignment μ propositionally satisfying φ . This is done recursively, according to the following steps:

- (base) If $\varphi = \top$, then μ satisfies φ . Thus, if μ is K_m -satisfiable, then φ is K_m -satisfiable. Therefore KSAT_F invokes KSAT_A(μ) (where “A” stands for Assignment), which returns a truth value asserting whether μ is K_m -satisfiable or not.
- (backtrack) If $\varphi = \perp$, then μ does not satisfy φ , so that KSAT_F returns *False*.
- (unit) If a literal l occurs in φ as a unit clause, then l must be assigned \top .⁷ To obtain this, KSAT_F is invoked recursively with arguments the formula returned by *assign*(l , φ) and the assignment obtained by adding l to μ .

⁷A notion of unit clause for non-CNF propositional formulas is given in [AG93].

$assign(l, \varphi)$ substitutes every occurrence of l in φ with \top and evaluates the result.

- (split) If none of the above situations occurs, then *choose-literal*(φ) returns an unassigned literal l according to some heuristic criterion. Then $KSAT_F$ is first invoked recursively with arguments $assign(l, \varphi)$ and $\mu \wedge l$. If the result is negative, then $KSAT_F$ is invoked with arguments $assign(\neg l, \varphi)$ and $\mu \wedge \neg l$.

$KSAT_F$ is a variant of the “classic” DPLL algorithm [DP60, DLL62]. The $KSAT_F$ schema differs from that of classic DPLL by only two steps.

The first difference is the “base” case: when finding an assignment μ which propositionally satisfies the input formula, it simply returns “True”. $KSAT_F$ instead is supposed also to check the K_m -satisfiability of the corresponding set of literals, by invoking $KSAT_A$ on μ . If the latter returns *true*, then the whole formula is satisfiable and $KSAT_F$ returns *True* as well; otherwise, $KSAT_F$ backtracks and looks for the next assignment.

The second difference is in the fact that in $KSAT_F$ the pure-literal step [DLL62] is removed.⁸ In fact the sets of assignments generated by DPLL with pure-literal might be incomplete and might cause incorrect results, as shown by the following example.

Example 3. Let φ be the following formula:

$$(\Box_1 A_1 \vee A_1) \wedge (\Box_1(A_1 \rightarrow A_2) \vee A_2) \wedge (\neg \Box_1 A_2 \vee A_2) \wedge (\neg A_2 \vee A_3) \wedge (\neg A_2 \vee \neg A_3).$$

φ is K_m -satisfiable, because $\mu = \{A_1, \neg A_2, \Box_1(A_1 \rightarrow A_2), \neg \Box_1 A_2\}$ is an assignment which propositionally satisfies φ and which is also modally consistent. It is easy to see that no satisfiable assignment propositionally satisfying φ assigns $\Box_1 A_1$ to true. As $\Box_1 A_1$ occurs only positively in φ , DPLL with the pure literal rule would assign $\Box_1 A_1$ to true as first step, which would lead the procedure to return *False*.

With these simple modifications, the embedded DPLL procedure works as an enumerator of a complete set of assignments, whose K_m -satisfiability is recursively checked by $KSAT_A$.

$KSAT_A(\mu)$ invokes $KSAT_{AR}(\mu_r)$ (where “ AR ” stands for *Restricted Assignment*) for every box index r . This is repeated until either $KSAT_{AR}$ returns a negative value (in which case $KSAT_A(\mu)$ returns *False*) or no more \Box_r ’s are available (in which case $KSAT_A(\mu)$ returns *True*). $KSAT_{AR}(\mu_r)$ invokes $KSAT(\varphi^{rj})$ for any conjunct $\neg \Box_r \beta_{rj}$ occurring in μ_r . Again, this is repeated until either $KSAT$ returns a negative value (in which case $KSAT_{AR}(\mu_r)$ returns *False*) or no more $\neg \Box_r \beta_{rj}$ ’s are available (in which case $KSAT_{AR}(\mu_r)$ returns *True*). Notice that $KSAT_F$, $KSAT_A$ and $KSAT_{AR}$ are a direct implementation of propositions 1, 3 and 4, respectively. This guarantees their correctness and completeness.

32.3.5. Modal DPLL vs. Modal Tableaux

[GS96a, GS96b, GS00, GGST98, GGST00, HPS99, HPSS00] presented extensive

⁸Alternatively, the application of the pure-literal rule can be restricted to atomic propositions only.

empirical comparisons, in which DPLL-based procedures outperformed tableau-based ones, with performance gaps that can reach orders of magnitude. (Similar performance gaps between tableau-based vs. DPLL-based procedures were obtained lately also in a completely-different context [ACG00].) Remarkably, most such results were obtained with tools implementing the “classic” DPLL procedure of §32.3.4, very far from the efficiency of current DPLL implementations.

We concentrate on the basic tableau-based and DPLL-based algorithms for K_m -satisfiability described in §32.3.2 and §32.3.4. Both procedures work (i) by enumerating truth assignments which propositionally satisfy the input formula φ and (ii) by recursively checking the K_m -satisfiability of the assignments found. Both algorithms perform the latter step in the same way. The key difference is thus in the way they handle propositional inference. [GS96b, GS00] remarked that, regardless the quality of implementation and the optimizations performed, tableau-based procedures have, with respect to DPLL-based procedures, two weaknesses which make them intrinsically less efficient, and whose effects get up to exponentially amplified when using them in modal inference. We consider them in turn.

Syntactic vs. semantic branching. In a propositional tableau truth assignments are generated as branches induced by the application of the \vee -rule to *disjunctive subformulas* of the input formula φ . Thus, they perform what we call *syntactic branching* [GS96b], that is, the branching in the search tree is induced by the syntactic structure of φ . As discussed in [D’A92, DM94], an application of the \vee -rule generates two subtrees which can be *mutually consistent*, i.e., which may share propositional models.⁹ Therefore, the set of truth assignments enumerated by propositional tableau procedures grows exponentially with the number of disjunctions occurring positively in φ , regardless the fact that it may contain up to exponentially-many duplicated and/or subsumed assignments.

Things get even worse in the modal case. When testing K_m -satisfiability, unlike the propositional case where tableaux look for *one* assignment satisfying the input formula, the propositional tableaux are used to enumerate *all* the truth assignments, which must be recursively checked for K_m -consistency. This requires checking recursively possibly-many sub-formulas of the form $\bigwedge_i \alpha_{ri} \wedge \neg\beta_j$ of depth $d - 1$, for which a propositional tableau will enumerate all truth assignments, and so on. At all levels of nesting, a redundant truth assignment introduces a redundant modal search tree. Thus, with modal formulas the redundancy of the propositional case propagates with the modal depth, and, in the worst case, the number of redundant truth assignments can become exponential.

DPLL instead, performs a search which is based on what we call *semantic branching* [GS96b], that is, a branching on the *truth value* of sub-formulas ψ of φ

⁹As pointed out in [D’A92, DM94], the propositional tableaux rules are unable to represent *bivalence*: “every proposition is either true or false, *tertium non datur*”. This is a consequence of the elimination of the cut rule in cut-free sequent calculi, from which propositional tableaux are derived.

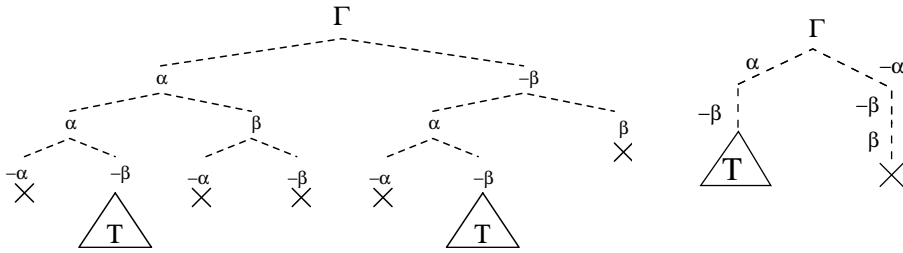


Figure 32.3. Search trees for the formula $\Gamma = (\alpha \vee \neg\beta) \wedge (\alpha \vee \beta) \wedge (\neg\alpha \vee \neg\beta)$. Left: a tableau-based procedure. Right: a DPLL-based procedure.

(typically atoms): ¹⁰

$$\frac{\varphi}{\varphi[\psi/\top] \quad \varphi[\psi/\perp]},$$

where $\varphi[\psi/\top]$ is the result of substituting with \top all occurrences of ψ in φ and then simplify the result. Thus, every branching step generates two *mutually-inconsistent* subtrees. ¹¹ Because of this, DPLL always generates non-redundant sets of assignments. This avoids any search duplication and, in the case of modal search, any recursive exponential propagation of such a redundancy.

Example 4. Consider the simple formula $\Gamma = (\alpha \vee \neg\beta) \wedge (\alpha \vee \beta) \wedge (\neg\alpha \vee \neg\beta)$, where α and β are modal atoms s.t. $\alpha \wedge \neg\beta$ is not modally consistent. and let d be the depth of Γ . The only possible assignment propositionally satisfying Γ is $\mu = \alpha \wedge \neg\beta$. Look at Figure 32.3 left. Assume that in a tableau-based procedure, the \vee -rule is applied to the three clauses occurring in Γ in the order they are listed. Then two distinct but identical open branches are generated, both representing the assignment μ . Then the tableau expands the two open branches in the same way, until it generates two identical (and possibly big) closed modal sub-trees T of modal depth d , each proving the K_m -unsatisfiability of μ .

This phenomenon may repeat itself at the lower level in each sub-tree T , and so on. For instance, if $\alpha = \Box_1((\alpha' \vee \neg\beta') \wedge (\alpha' \vee \beta'))$ and $\beta = \Box_1(\alpha' \wedge \beta')$, then at the lower level we have a formula Γ' of depth $d - 1$ analogous to Γ . This propagates exponentially the redundancy with the depth d .

Finally, notice that if we considered the formula $\Gamma^K = \bigwedge_{i=1}^K (\alpha_i \vee \neg\beta_i) \wedge (\alpha_i \vee \beta_i) \wedge (\neg\alpha_i \vee \neg\beta_i)$, the tableau would generate 2^K identical truth assignments $\mu^K = \bigwedge_i \alpha_i \wedge \neg\beta_i$, and things would get exponentially worse.

Look at Figure 32.3, right. A DPLL-based procedure branches asserting $\alpha = \top$ or $\alpha = \perp$. The first branch generates $\alpha \wedge \neg\beta$, while the second gives $\neg\alpha \wedge \neg\beta \wedge \beta$, which immediately closes. Therefore, only one instance of $\mu = \alpha \wedge \neg\beta$ is generated. The same applies to μ^K .

¹⁰Notice that the notion of “semantic branching” introduced in [GS96b] is stronger than that lately used in [Hor98b, HPS99], the former corresponding to the latter plus the usage of unit-propagation.

¹¹This fact holds for both “classic” [DP60, DLL62] and “modern” DPLL (see, e.g., [ZM02]), because in both cases two branches differ for the truth value of at least one atom, although for the latter case the explanation is slightly more complicate.

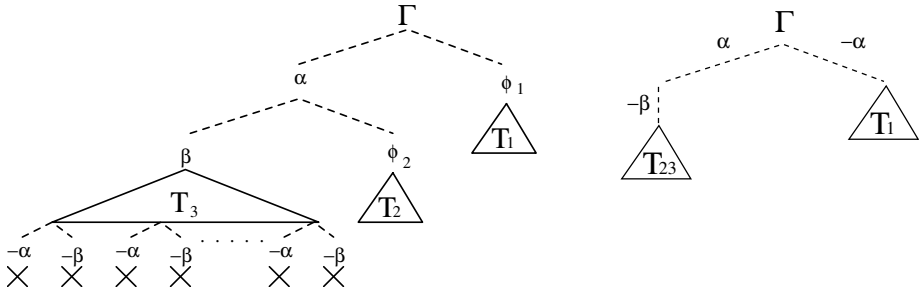


Figure 32.4. Search trees for the formula $\Gamma = (\alpha \vee \phi_1) \wedge (\beta \vee \phi_2) \wedge \phi_3 \wedge (\neg\alpha \vee \neg\beta)$. Left: a tableau-based procedure. Right: a DPLL-based procedure.

Detecting constraint violations. A propositional formula φ can be seen as a set of constraints for the truth assignments which possibly satisfy it. For instance, a clause $A_1 \vee A_2$ constrains every assignment not to set both A_1 and A_2 to \perp . Unlike tableaux, DPLL prunes a branch as soon as it violates some constraint of the input formula. (For instance, in KSAT this is done by the function *assign*.)

Example 5. Consider the formula $\Gamma = (\alpha \vee \phi_1) \wedge (\beta \vee \phi_2) \wedge \phi_3 \wedge (\neg\alpha \vee \neg\beta)$, α and β being atoms, ϕ_1 , ϕ_2 and ϕ_3 being sub-formulas, such that $\alpha \wedge \beta \wedge \phi_3$ is propositionally satisfiable and $\alpha \wedge \phi_2$ is K_m -unsatisfiable. Look at Figure 32.4, left. Again, assume that, in a tableau-based procedure, the \vee -rule is applied in order, left to right. After two steps, the branch α, β is generated, which violates the constraint imposed by the last clause $(\neg\alpha \vee \neg\beta)$. A tableau-based procedure is not able to detect such a violation until it explicitly branches on that clause, that is, only after having generated the whole sub-tableau T_3 for $\alpha \wedge \beta \wedge \phi_3$, which may be rather big. DPLL instead (Figure 32.4, right) avoids generating the violating assignment detects the violation and immediately prunes the branch.

32.4. Advanced Modal DPLL

In this section we present the most important optimizations of the DPLL-based procedures described in §32.3, and one extension to non-normal modal logics.

32.4.1. Optimizations

As described in §32.3.4, the first DPLL-based tools of [GS96a, GS96b, SV98, GS00] were based on the “classic” recursive DPLL schema [DP60, DLL62]. Drastic improvements in performances were lately obtained by importing ideas and techniques from the SAT literature and/or by directly implementing tools on top of modern DPLL solvers, which are applied to the Boolean abstraction of the input formula [GGST98, GGST00, HPS99, Tac99, GGT01, HM01].

In particular, modern DPLL implementation are non-recursive, and are based on very efficient, destructive data structures to handle Boolean formulas and assignments. They benefit of sophisticated search techniques (e.g., backjumping, learning, restarts [MSS96, BS97, GSK98]), smart splitting heuristics (e.g.,

[MMZ⁺01, GN02, ES04]), highly-engineered data structures and implementation tricks (e.g., the two-watched literal scheme [MMZ⁺01]), and advanced preprocessing techniques [Bra01, BW03, EB05]. In particular, modern DPLL implementations perform *conflict analysis* on failed assignments μ 's, which detect the *reason* of each failure, that is, a (typically much smaller) subset μ' of μ which alone causes the failure. When this happens, the procedure

- adds the negation of μ' as a new clause to the formula, so that no assignment containing μ' will be ever investigated again. This technique is called *learning*;
- backtracks to the highest point in the stack where one literal l in the learned clause $\neg\mu'$ is not assigned, it unit propagates l , and it proceeds with the search. This technique is called *backjumping*.

Backjumping and learning are of great interest in our discussion, as it will be made clear in §32.4.1.4. The other DPLL optimizations come for free by using state-of-the-art SAT solvers and they are substantially orthogonal to our discussion, so that they will not be discussed here.

We describe some further optimizations which have been proposed to the basic schema of §32.3.4. For better readability, the description will refer to the case of K_m , but they can be extended to other logics. Most of these techniques and optimizations have lately been adopted by the so-called lazy tools for Satisfiability Modulo Theories, SMT (see §26.4.3).

32.4.1.1. Normalizing atoms

One potential source of inefficiency for DPLL-based procedures is the occurrence in the input formula of equivalent though syntactically-different atoms (e.g., $\Box_r(A_1 \vee A_2)$ and $\Box_r(A_2 \vee A_1)$), or pairs atoms in which one is equivalent to the negation of the other (e.g. $\Box_r(A_1 \vee A_2)$ and $\Diamond_r(\neg A_1 \wedge \neg A_2)$). If two atoms ψ_1, ψ_2 are s.t. $\psi_1 \neq \psi_2$ and $\models \psi_1 \leftrightarrow \psi_2$ [resp. $\psi_1 \neq \neg\psi_2$ and $\models \psi_1 \leftrightarrow \neg\psi_2$], then they are recognized as distinct Boolean atoms $B_1 =_{def} \mathcal{L}2\mathcal{P}(\psi_1)$ and $B_2 =_{def} \mathcal{L}2\mathcal{P}(\psi_2)$, which may be assigned different [resp. identical] truth values by DPLL. This may cause the useless generation of many unsatisfiable assignments and the corresponding useless calls to KSAT_A (e.g., up to $2^{|Atoms^0(\varphi)|-2}$ calls on assignments like $\{\Box_r(A_1 \vee A_2), \neg\Box_r(A_2 \vee A_1)\dots\}$).

In order to avoid these problems, it is wise to preprocess atoms so that to map as many as possible equivalent literals into syntactically identical ones [GS96a, GS00, HPS99]. This can be achieved by applying some rewriting rules, like, e.g.:

- *Drop dual operators:* $\Box_r(\varphi_1 \vee \varphi_2), \Diamond_r(\neg\varphi_1 \wedge \neg\varphi_2) \implies \Box_r(\varphi_1 \vee \varphi_2), \neg\Box_r(\varphi_1 \vee \varphi_2)$, or even $(\varphi_1 \wedge \varphi_2), (\neg\varphi_1 \vee \neg\varphi_2) \implies (\varphi_1 \wedge \varphi_2), \neg(\varphi_1 \wedge \varphi_2)$
- *Exploit associativity:* $\Box_r(\varphi_1 \vee (\varphi_2 \vee \varphi_3)), \Box_r((\varphi_1 \vee \varphi_2) \vee \varphi_3) \implies \Box_r(\varphi_1 \vee \varphi_2 \vee \varphi_3)$,
- *Sort:* $\Box_r(\varphi_2 \vee \varphi_1 \vee \varphi_3), \Box_r(\varphi_3 \vee \varphi_1 \vee \varphi_2) \implies \Box_r(\varphi_1 \vee \varphi_2 \vee \varphi_3)$.
- *Exploit properties of normal modal logics:* $\Box_r(\varphi_1 \wedge \varphi_2) \implies \Box_r\varphi_1 \wedge \Box_r\varphi_2$ if $\mathcal{L} \in \mathcal{N}$.
- *Exploit specific properties of some logic \mathcal{L} :* $\Box_r\Box_r\varphi_1 \implies \Box_r\varphi_1$ if \mathcal{L} is S5.

Notice that pre-conversion to BNF (§32.2.1) goes in this direction.

Example 6. Consider the modal atoms occurring in the formula φ in Example 1. For every modal atom in φ there are $3! = 6$ equivalent permutations, which are all mapped into one atom if the modal atoms are sorted. E.g., if we consider an equivalent formula φ' in which the second occurrence of the atom $\Box_2(\neg A_4 \vee A_5 \vee A_2)$, occurring in rows 3 and 5, is rewritten as $\Box_2(A_5 \vee \neg A_4 \vee A_2)$, then the latter will be encoded by $\mathcal{L2P}$ into a different Boolean variable, namely B_9 , which could be assigned by DPLL a different truth value wrt. B_3 , generating an modally inconsistent assignment μ' . If all atoms in φ' are pre-sorted, then the problem does not occur.

32.4.1.2. Early Pruning

Another optimization [GS96a, GS00, Tac99] was conceived after the empirical observation that most assignments found by DPLL are “trivially” K_m -unsatisfiable, that is, they will remain K_m -unsatisfiable even after removing some of their conjuncts. If an incomplete ¹² assignment μ' is K_m -unsatisfiable, then all its extensions are K_m -unsatisfiable. If the unsatisfiability of μ' is detected on time, then this prevents checking the K_m -satisfiability of all the up to $2^{|Atoms^0(\varphi)| - |\mu'|}$ truth assignments which extend μ' .

This suggests the introduction of an intermediate K_m -satisfiability test on incomplete assignments just before the split. (Notice there is no need to introduce similar tests before unit propagation.) In the basic algorithm of Figure 32.2, this is done by introducing the three lines below in the function KSAT_F of Figure 32.2, just before the “split”:

```

if (Likely-Unsatisfiable( $\mu$ )           /* early-pruning */
    if not  $\text{KSAT}_A(\mu)$ 
    then return False;
```

(We temporarily ignore the test performed by *Likely-Unsatisfiable*.) KSAT_A is invoked on the current incomplete assignment μ . If $\text{KSAT}_A(\mu)$ returns *False*, then all possible extensions of μ are unsatisfiable, and therefore KSAT_F returns *False*. The introduction of this intermediate check, which is called *early pruning*, caused a drastic improvement in the overall performances [GS96a, GS00].

Example 7. Consider the formula φ of Example 1. Suppose that, after three recursive calls, KSAT_F builds the incomplete assignment:

$$\mu' = \Box_1(\neg A_1 \vee A_4 \vee A_3) \wedge \Box_1(\neg A_2 \vee A_1 \vee A_4) \wedge \neg \Box_1(A_4 \vee \neg A_2 \vee A_3)$$

(rows 6, 7 and 4 of φ). If it is invoked on μ' , KSAT_A will check the K_2 -satisfiability of the formula

$$(\neg A_1 \vee A_4 \vee A_3) \wedge (\neg A_2 \vee A_1 \vee A_4) \wedge \neg A_4 \wedge A_2 \wedge \neg A_3,$$

which is unsatisfiable. Therefore there will be no more need to select further literals, and KSAT_F will backtrack.

¹²By *incomplete* assignment μ for φ we mean that μ has not assigned enough atoms to determine whether $\mu \models \varphi$ or not.

The intermediate consistency checks, however, may introduce some useless calls to KSAT_A .

One way of addressing this problem is to condition the calls to KSAT_A in early-pruning steps to some heuristic criteria (here represented by the heuristic function *Likely-Unsatisfiable*). The main idea is to avoid invoking KSAT_A when it is very unlikely that, since the last call, the new literals added to μ can cause inconsistency: e.g., when they are added only literals which are purely-propositional or contain new Boolean atoms [GS96a, GS00].

Another way is to make KSAT_A work in an *incremental* way: if for some box index $r \in \{1 \dots m\}$ no literal of the form $\Box_r \psi$ or $\neg \Box_r \psi$ has been added to μ since the last call to KSAT_A , then KSAT_A can avoid performing the corresponding call to KSAT_{AR} ; moreover, if for some box index $r \in \{1 \dots m\}$ no positive $\Box_r \psi$'s have been added to μ since the last call to KSAT_A , then KSAT_{AR} can avoid calling recursively KSAT on the subformulas $(\bigwedge_i \alpha_{ri} \wedge \neg \beta_{rj})$ s.t. $\neg \Box_r \beta_{rj}$ was already passed to KSAT_A in the last call [Tac99].

32.4.1.3. Caching

This section is an overview of [GT01] to which we refer for further reading. Consider the basic version of KSAT algorithm in Figure 32.2. Without loss of generality, in the remainder of this section we assume that $|\mathcal{B}| = 1$, so that the call to KSAT_A is the same as KSAT_{AR} . The extension to the case where $|\mathcal{B}| > 1$ is straightforward since it simply requires checking the different modalities in separate calls to KSAT_{AR} . Given two assignments μ and μ' , it may be the case that $\text{KSAT}_A(\mu)$ and $\text{KSAT}_A(\mu')$ perform some equal substests, i.e., recursive calls to KSAT . This is the case, e.g., when μ and μ' differ only for the propositional conjuncts and there is at least one conjunct of the form $\neg \Box \beta$. To prevent recomputation, the obvious solution is to cache both the formula whose satisfiability is being checked and the result of the check. Then, the cache is consulted before performing each subtest to determine whether the result of the subtest can be assessed on the basis of the cache contents.

In the following, we assume to have two different caching mechanisms, each using a separate caching structure:

- S-cache to store and query about satisfiable formulas, and
- U-cache to store and query about unsatisfiable formulas.

In this way, storing a subtest amounts to storing the formula in the appropriate caching structure. Of course, the issue is how to implement effective caching mechanisms allowing to reduce the number of substests as much as possible. To this extent, the following considerations are in order:

1. if a formula $\bigwedge_{\alpha \in \Delta} \alpha \wedge \neg \beta$ has already been determined to be satisfiable then, if $\Delta \subseteq \Delta'$, we can conclude that also $\bigwedge_{\alpha \in \Delta} \alpha \wedge \neg \beta$ is satisfiable, and
2. if a formula $\bigwedge_{\alpha \in \Delta} \alpha \wedge \neg \beta$ has already been determined to be unsatisfiable then, if $\Delta \supseteq \Delta'$, we can conclude that also $\bigwedge_{\alpha \in \Delta} \alpha \wedge \neg \beta$ is unsatisfiable.

The above observations suggest the usage of caching mechanisms that allow for storing sets of formulas and for efficiently querying about subsets or supersets.

```

function KSATA(μ)
  Δ := {α | □α is a conjunct of μ};
  Γ := {β | ¬□β is a conjunct of μ};
  if U-cache.get(Δ, Γ) return False;
  Γr := S-cache.get(Δ, Γ);
  Γs := ∅;
  foreach β ∈ Γr do
    if not KSAT(∧α∈Δ α ∧ ¬β) then
      if Γs ≠ ∅ then S-cache.store(Δ, Γs);
      U-cache.store(Δ, β);
      return False
    else Γs := Γs ∪ {β};
  S-cache.store(Δ, Γs);
  return True.

```

Figure 32.5. KSAT_A: satisfiability checking for K with caching

In other words, given a subtest

$$\text{KSAT}\left(\bigwedge_{\alpha \in \Delta} \alpha \wedge \neg\beta\right), \quad (32.19)$$

we want to be able to query our S-cache about the presence of a formula

$$\bigwedge_{\alpha \in \Delta'} \alpha \wedge \neg\beta \quad (32.20)$$

with $\Delta \subseteq \Delta'$ (query for subsets or *subset-matching*). Analogously, given the subtest (32.19), we want to be able to query our U-cache about the presence of a formula (32.20) with $\Delta \supseteq \Delta'$ (query for supersets or *superset-matching*). In this way, caching a subtest avoids the recomputation of the very same subtest, *and* of the possibly many “subsumed” subtests.

Observations 1 and 2 are independent of the particular modal logic being considered. They are to be taken into account when designing caching structures for satisfiability in any modal logic. Of course, depending on the particular modal logic considered, some other considerations might be in order. For example, in K, we observe that in KSAT_A there is a natural unbalance between satisfiable subtests and unsatisfiable ones. In fact, with reference to Figure 32.2, when testing an assignment μ

3. many subtests can be determined to be satisfiable, all sharing the same set Δ , and
4. at most one subtest may turn out to be unsatisfiable.

Observation 3 suggests that S-cache should be able to store satisfiable subtests sharing a common set Δ in a compact way. Therefore, S-cache associates the set Δ to the set $\Gamma' \subseteq \Gamma$, representing the “computed” satisfiable subtests $\bigwedge_{\alpha \in \Delta} \alpha \wedge \neg\beta$ for each $\beta \in \Gamma'$. Observation 4 suggests that U-cache should not care about subtests sharing a common Δ . Therefore, U-cache associates Δ to the single β for which the subtest $\bigwedge_{\alpha \in \Delta} \alpha \wedge \neg\beta$ failed.

Given the design issues outlined above, we can modify KSAT to yield the procedure KSAT_A shown in Figure 32.5. In the Figure:

- $\text{U-cache.get}(\Delta, \Gamma)$ returns *True* if U-cache contains a set Δ' such that $\Delta \supseteq \Delta'$, Δ' is associated with β and $\beta \in \Gamma$;
- $\text{S-cache.get}(\Delta, \Gamma)$ returns the set $\Gamma \setminus \Gamma'$ where Γ' is the union over all the sets Γ'' such that for some set $\Delta' \supseteq \Delta$, Γ'' is associated to Δ' in S-cache.
- $\text{U-cache.store}(\Delta, \beta)$ stores in U-cache the set Δ and associates β to it;
- $\text{S-cache.store}(\Delta, \Gamma)$ stores in S-cache the set Δ and associates to it the set Γ .

The new issue is now to implement effective data structures for S-cache and U-cache supporting the above functions. Clearly, we expect that the computational costs associated to the above functions will be superior to the computational costs associated to other caching structures designed for “equality-matching”, i.e., effectively supporting the functions obtained from the above by substituting “ \supseteq ” with “ $=$ ”. There is indeed a trade-off between “smart but expensive” and “simple but efficient” data-structures for caching. Of course, depending on

- the particular logic being considered, and
- the characteristics of the particular formula being tested,

we expect that one caching mechanism will lead to a faster decision process than the others.

Independently from the data-structure being used, the following (last) observation needs to be taken into account when dealing with modal logics whose decision problem is not in NP (e.g., K, S4):

5. testing the consistency of a formula may require an exponential number of substests.

This is the case for the Halpern and Moses formulas presented in [HM85] for various modal logics. Observation 5 suggests that it may be necessary to bound the size of the cache, and introduce mechanisms for deciding which formulas to discard when the bound is reached. Further discussion about the implementation of caching and low-level optimizations can be found in [GT01].

32.4.1.4. Modal Backjumping

Another very important optimization, called *modal backjumping* [Hor98a, PS98], generalizes the idea of backjumping in DPLL. KSAT_A can be easily modified so that, when invoked on a K_m -unsatisfiable set of modal literals μ , it returns also the subset μ' of μ which caused the inconsistency of μ . We call μ' , a *modal conflict set* of μ .

An easy way of computing μ' is that of returning the set $\mathcal{L2P}(\{\Box_r \alpha_{ri}\}_i \cup \{\neg \Box_r \beta_{rj}\})$ corresponding to the first formula $\varphi^{rj} = \bigwedge_i \alpha_{ri} \wedge \neg \beta_{rj}$ which is found unsatisfiable by KSAT.

Example 8. Consider the formula φ of Example 1. The assignment $\mu^P = \{B_6, B_8, B_2, \neg B_1, \neg B_5, B_3\}$ is found by KSAT_F , which satisfies $\mathcal{L2P}(\varphi)$. Thus

KSAT_A is given as input

$$\begin{aligned} \mu = & \quad \Box_1(\neg A_5 \vee A_4 \vee A_3) \wedge & & [\bigwedge_i \Box_1 \alpha_{1i}] \\ & \Box_1(\neg A_2 \vee A_1 \vee A_4) \wedge \Box_1(\neg A_2 \vee A_4 \vee A_5) \wedge & & [\bigwedge_j \neg \Box_1 \beta_{1j}] \\ & \neg \Box_1(\neg A_3 \vee \neg A_1 \vee A_2) \wedge \neg \Box_1(A_4 \vee \neg A_2 \vee A_3) \wedge & & \\ & \Box_2(\neg A_4 \vee A_5 \vee A_2) & & [\bigwedge_i \Box_2 \alpha_{2i}] \end{aligned}$$

and hence invokes KSAT_{AR} on the two restricted assignments:

$$\begin{aligned} \mu^1 = & \quad \Box_1(\neg A_5 \vee A_4 \vee A_3) \wedge & & [\bigwedge_i \Box_1 \alpha_{1i}] \\ & \Box_1(\neg A_2 \vee A_1 \vee A_4) \wedge \Box_1(\neg A_2 \vee A_4 \vee A_5) \wedge & & [\bigwedge_j \neg \Box_1 \beta_{1j}] \\ & \neg \Box_1(\neg A_3 \vee \neg A_1 \vee A_2) \wedge \neg \Box_1(A_4 \vee \neg A_2 \vee A_3) & & \\ \mu^2 = & \quad \Box_2(\neg A_4 \vee A_5 \vee A_2) & & [\bigwedge_i \Box_2 \alpha_{2i}]. \end{aligned}$$

μ^2 is trivially K_m -satisfiable. μ^1 requires invoking KSAT on the two formulas

$$\begin{aligned} \varphi^{11} = & (\neg A_5 \vee A_4 \vee A_3) \wedge (\neg A_2 \vee A_1 \vee A_4) \wedge \\ & (\neg A_2 \vee A_4 \vee A_5) \wedge A_3 \wedge A_1 \wedge \neg A_2, \\ \varphi^{12} = & (\neg A_5 \vee A_4 \vee A_3) \wedge (\neg A_2 \vee A_1 \vee A_4) \wedge \\ & (\neg A_2 \vee A_4 \vee A_5) \wedge \neg A_4 \wedge A_2 \wedge \neg A_3. \end{aligned}$$

The latter is unsatisfiable, from which we can conclude that

$$\Box_1(\neg A_5 \vee A_4 \vee A_3) \wedge \Box_1(\neg A_2 \vee A_1 \vee A_4) \wedge \Box_1(\neg A_2 \vee A_4 \vee A_5) \wedge \neg \Box_1(A_4 \vee \neg A_2 \vee A_3)$$

is K_m -unsatisfiable, so that $\{B_6, B_8, B_2, \neg B_5\}$ is a conflict set of μ^p .

The conflict set μ' found is then used to drive the backjumping mechanism of DPLL. Different strategies are possible. The DPLL-based modal tools [Hor98a, PS98] and earlier SMT tools [WW99] used to jump up to the most recent branching point s.t. at least one literal $l^p \in \mu'$ is not assigned. Intuitively, all open subbranches departing from the current branch at a lower decision point contain μ' , so that there is no need to explore them; this allows for pruning all these subbranches from the search tree. (Notice that these strategies do not explicitly require adding the clause $\neg \mu'$ to φ .) More sophisticated versions of this technique, which mirror the most-modern backjumping techniques introduced in DPLL, were lately introduced in the context of SMT (see §26.4.3.5).

In substance, modal backjumping differs from standard Boolean backjumping only for the notion of conflict set used: whilst a Boolean conflict set μ is an assignment which causes a propositional inconsistency if conjoined to φ (i.e. s.t. $\mu \wedge \varphi \models_p \perp$), a modal conflict set is a set of literals which in K_m -inconsistent (i.e. s.t. $\mu \models \perp$).

32.4.1.5. Pure-literal filtering

This technique, which we call *pure-literal filtering*,¹³ was implicitly proposed by [WW99] and then generalized and adopted in the *SAT tool [Tac99] (and lately imported into SMT [ABC⁺02], see §26.4.3.7). The idea is that, if we have non-Boolean atoms occurring only positively [resp. negatively] in the input formula,

¹³Also called *triggering* in [WW99, ABC⁺02].

we can safely drop every negative [resp. positive] occurrence of them from the assignment μ to be checked by KSAT_A . (The correctness and completeness of this process is a consequence of proposition 2 in §32.3.1.)

There are some benefits for this behavior. Let μ' be the reduced version of μ .

First, μ' might be K_m -satisfiable despite μ is K_m -unsatisfiable. If so, and if μ (and hence μ') propositionally satisfies φ , then KSAT_F can stop, potentially saving a lot of search.

Second, if both μ' and μ are K_m -unsatisfiable, the call to KSAT_A on μ' rather than that on μ can cause smaller conflict sets, in order to improve the effectiveness of backjumping and learning.

Third, checking the K_m -satisfiability of μ' rather than that of μ can be significantly faster. In fact, suppose $\Box_r\beta_{rj}$ occurs only positively in φ and it is assigned a negative value by KSAT_F , so that $\neg\Box_r\beta_{rj} \in \mu$ but $\neg\Box_r\beta_{rj} \notin \mu'$. Thus $\neg\Box_r\beta_{rj}$ will not occur in the restricted assignment μ' fed to KSAT_{AR} , avoiding the call to KSAT on $(\bigwedge_i \alpha_{ri} \wedge \neg\beta_{rj})$. This allows for extending the notion of “incrementality” of §32.4.1.2, by considering only the literals in μ' rather than those in μ .

32.4.2. Extensions to Non-Normal Modal Logics

This section briefly surveys some of the contents of [GGT01] to which we refer for further reading. Following the notation of [GGT01], we say that an assignment μ satisfies a formula φ if μ entails φ by propositional reasoning, and that a formula φ is *consistent* in a logic L (or *L-consistent*) if $\neg\varphi$ is not a theorem of L, i.e., if $\neg\varphi \notin L$. Whether an assignment is consistent, depends on the particular classical modal logic L being considered. Furthermore, depending on the logic L considered, the consistency problem for L (i.e., determining whether a formula is consistent in L) belongs to different complexity classes. In particular, the consistency problem for E, EM, EN, EMN is NP-complete, while for EC, ECN, EMC it is PSPACE-complete (see [Var89, FHMV95]). Here, to save space, we divide these eight logics in two groups. We present the algorithms for checking the L-consistency of an assignment first in the case in which L is one of E, EM, EN, EMN, and then in the case in which L is one of the others.

32.4.2.1. Logics E, EM, EN, EMN

The following proposition is an easy consequence of the results presented in [Var89].

Proposition 5. *Let $\mu = \bigwedge_i \Box\alpha_i \wedge \bigwedge_j \neg\Box\beta_j \wedge \gamma$ be an assignment in which γ is a propositional formula. Let L be one of the logics E, EM, EN, EMN. μ is consistent in L if for each conjunct $\neg\Box\beta_j$ in μ one of the following conditions is satisfied:*

- $(\alpha_i \equiv \neg\beta_j)$ is L-consistent for each conjunct $\Box\alpha_i$ in μ , and $L=E$;
- $(\alpha_i \wedge \neg\beta_j)$ is L-consistent for each conjunct $\Box\alpha_i$ in μ , and $L=EM$;
- $\neg\beta_j$ and $(\alpha_i \equiv \neg\beta_j)$ are L-consistent for each conjunct $\Box\alpha_i$ in μ , and $L=EN$;
- $\neg\beta_j$ and $(\alpha_i \wedge \neg\beta_j)$ are L-consistent for each conjunct $\Box\alpha_i$ in μ , and $L=EMN$.

```

function LSATA(μ)
  foreach conjunct □βj do
    foreach conjunct □αi do
      if M[i, j] = Undef then M[i, j] := LSAT(αi ∧ ¬βj);
      if L ∈ {EN, EMN} and M[i, j] = True then M[j, j] := True;
      if L ∈ {E, EN} and M[i, j] = False then
        if M[j, i] = Undef then M[j, i] := LSAT(¬αi ∧ βj);
        if L = EN and M[j, i] = True then M[i, i] := True;
        if M[j, i] = False then return False
      end
    if L ∈ {EN, EMN} then
      if M[j, j] = Undef then M[j, j] := LSAT(¬βj);
      if M[j, j] = False then return False
    end;
  return True.

```

Figure 32.6. LSAT_A for E, EM, EN, EMN

When implementing the above conditions, care must be taken in order to avoid repetitions of consistency checks. In fact, while an exponential number of assignments satisfying the input formula can be generated, at most n^2 checks are possible in L, where n is the number of “□” in the input formula. Given this upper bound, for each new consistency check, we can cache the result for a future possible re-utilization in a $n \times n$ matrix M. This ensures that at most n^2 consistency checks will be performed. In more detail, given an enumeration $\varphi_1, \varphi_2, \dots, \varphi_n$ of the boxed subformulas of the input formula, $M[i, j]$, with $i \neq j$, stores the result of the consistency check for $(\varphi_i \wedge \neg\varphi_j)$. $M[i, i]$ stores the result of the consistency check for $\neg\varphi_i$. Initially, each element of the matrix M has value *Undef* (meaning that the corresponding test has not been done yet). The result is the procedure LSAT_A in Figure 32.6, where the procedure LSAT is identical to the procedure KSAT modulo the call to KSAT_A which must be replaced by LSAT_A.

Consider Figure 32.6 and assume that L=E or L=EN. Given a pair of conjuncts □α_i and ¬□β_j, we split the consistency test for $(\alpha_i \equiv \neg\beta_j)$ in two simpler sub-tests:

- first, we test whether $(\alpha_i \wedge \neg\beta_j)$ is consistent, and
- only if this test gives *False*, we test whether $(\neg\alpha_i \wedge \beta_j)$ is consistent.

Notice also that, in case L=EN or L=EMN, if we know that, e.g., $(\alpha_i \wedge \neg\beta_j)$ is consistent, then also $\neg\beta_j$ is consistent and we store this result in $M[j, j]$. The following proposition ensures the correctness of LSAT in the case of E, EM, EN and EMN.

Proposition 6. *Let $\mu = \bigwedge_i \square\alpha_i \wedge \bigwedge_j \neg\square\beta_j \wedge \gamma$ be an assignment in which γ is a propositional formula. Let L be one of the logics E, EM, EN, EMN. Assume that, for any formula φ whose depth is less than the depth of μ , LSAT(φ)*

- *returns True if φ is L-consistent, and*
- *False otherwise.*

```

function LSATA( $\bigwedge_i \Box \alpha_i \wedge \bigwedge_j \neg \Box \beta_j \wedge \gamma$ )
 $\Delta := \{\alpha_i \mid \Box \alpha_i \text{ is a conjunct of } \mu\};$ 
foreach conjunct  $\Box \beta_j$  do
   $\Delta' := \Delta;$ 
  if  $L \in \{EC, ECN\}$  then
    foreach conjunct  $\Box \alpha_i$  do
      if  $M[j, i] = \text{Undef}$  then  $M[j, i] := \text{LSAT}(\neg \alpha_i \wedge \beta_j);$ 
      if  $M[j, i] = \text{True}$  then  $\Delta' = \Delta' \setminus \{\alpha_i\}$ 
    end;
  if  $L \in \{ECN\}$  or  $\Delta' \neq \emptyset$  then
    if not  $\text{LSAT}(\bigwedge_{\alpha_i \in \Delta'} \alpha_i \wedge \neg \beta_j)$  then return False
  end;
return True.

```

Figure 32.7. LSAT_A for EC, ECN, EMC

LSAT_A(μ) returns True if μ is L-consistent, and False otherwise.

32.4.2.2. Logics EC, ECN, EMC

The following proposition is an easy consequence of the results presented in [Var89].

Proposition 7. Let $\mu = \bigwedge_i \Box \alpha_i \wedge \bigwedge_j \neg \Box \beta_j \wedge \gamma$ be an assignment in which γ is a propositional formula. Let Δ be the set of formulas α_i such that $\Box \alpha_i$ is a conjunct of μ . Let L be one of logics EC, ECN, EMC. μ is consistent in L if for each conjunct $\neg \Box \beta_j$ in μ one of the following conditions is satisfied:

- $((\bigwedge_{\alpha_i \in \Delta'} \alpha_i) \equiv \neg \beta_j)$ is L-consistent for each non empty subset Δ' of Δ , and $L=EC$;
- $((\bigwedge_{\alpha_i \in \Delta'} \alpha_i) \equiv \neg \beta_j)$ is L-consistent for each subset Δ' of Δ , and $L=ECN$;
- Δ is empty or $((\bigwedge_{\alpha_i \in \Delta} \alpha_i) \wedge \neg \beta_j)$ is L-consistent, and $L=EMC$;

Assume that $L=EC$ or $L=ECN$. The straightforward implementation of the corresponding condition may lead to an exponential number of checks in the cardinality $|\Delta|$ of Δ . More carefully, for each conjunct $\neg \Box \beta_j$ in μ , we can perform at most $|\Delta| + 1$ checks if

1. for each formula α_i in Δ , we first check whether $(\neg \alpha_i \wedge \beta_j)$ is consistent in L . Let Δ' be the set of formulas for which the above test fails. Then,
2. in case $L=ECN$ or $\Delta' \neq \emptyset$, we perform the last test, checking whether $((\bigwedge_{\alpha_i \in \Delta'} \alpha_i) \wedge \neg \beta_j)$ is consistent in L .

Furthermore, the result of the consistency checks performed in the first step can be cached in a matrix M analogous to the one used in the previous subsection.

If $L=EC$ or $L=ECN$, the procedure LSAT_A in Figure 32.7 implements the above ideas. Otherwise, it is a straightforward implementation of the conditions in proposition 7. The following proposition ensures the correctness of LSAT in the case of E, EM, EN and EMN.

Proposition 8. Let $\mu = \bigwedge_i \Box \alpha_i \wedge \bigwedge_j \neg \Box \beta_j \wedge \gamma$ be an assignment in which γ is a propositional formula. Let L be one of logics *EC*, *ECN*, *EMC*. Assume that, for any formula φ whose depth is less than the depth of μ , $Lsat(\varphi)$

- returns *True* if φ is L -consistent, and
- *False* otherwise.

$LSAT_A(\mu)$ returns *True* if μ is L -consistent, and *False* otherwise.

32.5. The OBDD-based Approach

In this section we briefly survey the basics of the OBDD-based approach to implement decision procedures for modal K , and we refer the reader to [PSV02, PV03] for further details. The contents of this section and the next borrow from [PSV06], including basic notation and the description of the algorithms.

The OBDD-based approach is inspired by the automata-theoretic approach for logics with the tree-model -property. In that approach, one proceeds in two steps. First, an input formula is translated to a tree automaton that accepts all the tree models of the formula. Second, the automaton is tested for non-emptiness, i.e., whether it accepts some tree. The approach described in [PSV02] combines the two steps and carries out the non-emptiness test without explicitly constructing the automaton. The logic K is simple enough that the automaton's non-emptiness test consists of a single fixpoint computation, which starts with a set of states and then repeatedly applies a monotone operator until a fixpoint is reached. In the automaton that corresponds to a formula, each state is a *type*, i.e., a set of formulas satisfying some consistency conditions. The algorithms that we describe here start from some set of types and then repeatedly apply a monotone operator until a fixpoint is reached.

32.5.1. Basics

To aid the description of the OBDD-based algorithms we introduce some additional notation. The set of propositional atoms used in a formula is denoted $AP(\varphi)$, and, given a formula ψ , we call its set of subformulas $\text{sub}(\psi)$. For $\varphi \in \text{sub}(\psi)$, we can define $\text{depth}(\varphi)$ in the usual way. If not stated otherwise, we assume all formulas to be in BNF. The *closure* of a formula $\text{cl}(\psi)$ is defined as the smallest set such that, for all subformulas φ of ψ , if φ is not of the form $\neg\varphi'$, then $\{\varphi, \neg\varphi\} \subseteq \text{cl}(\psi)$. The algorithms that we present here work on *types*, i.e., maximal sets of formulas that are consistent w.r.t. the Boolean operators, and where (negated) box formulas are treated as atoms. A set of formulas $a \subseteq \text{cl}(\psi)$ is called a ψ -*type* (or simply a type if ψ is clear from the context) if it satisfies the following conditions:

- If $\varphi = \neg\varphi'$, then $\varphi \in a$ iff $\varphi' \notin a$.
- If $\varphi = \varphi' \wedge \varphi''$, then $\varphi \in a$ iff $\varphi' \in a$ and $\varphi'' \in a$.
- If $\varphi = \varphi' \vee \varphi''$, then $\varphi \in a$ iff $\varphi' \in a$ or $\varphi'' \in a$.

For a set of types T , we define the maximal accessibility relation $\Delta \subseteq T \times T$ as follows.

$$\Delta(t, t') \text{ iff for all } \Box\varphi' \in t, \text{ we have } \varphi' \in t'$$

```

X := Init( $\psi$ )
repeat
  X' := X
  X := Update(X')
until X = X'
if exists  $x \in X$  such that  $\psi \in x$  then
  return " $\psi$  is satisfiable"
else
  return " $\psi$  is not satisfiable"

```

Figure 32.8. Basic schema for the OBDD-based algorithm.

In Figure 32.8 we present the basic schema for the OBDD-based decision procedures. The schema can be made to work in two fashions, called top-down and bottom-up in [PSV06], according to the definition of the accessory functions `Init` and `Update`. In both cases, since the algorithms operate with elements in a finite lattice $2^{\text{cl}(\psi)}$ and use a monotone `Update`, they are bound to terminate. In the case of the top-down approach, the accessory functions are defined as:

- `Init(ψ)` is the set of all ψ -types.
- `Update(T) := $T \setminus \text{bad}(T)$` , where `bad(T)` are the types in T that contain unwitnessed negated box formulas. More precisely,

$$\text{bad}(T) := \{t \in T \mid \text{there exists } \neg\Box\varphi \in t \text{ and,} \\ \text{for all } u \in T \text{ with } \Delta(t, u), \text{ we have } \varphi \in u\}.$$

Intuitively, the top-down algorithm starts with the set of *all* types and remove those types with “possibilities” $\Diamond\varphi$ for which no “witness” can be found. In the bottom-up approach, the accessory functions are defined as:

- `Init(ψ)` is the set of all those types that do not require any witness, which means that they do not contain any negated box formula, or equivalently, that they contain all positive box formulas in `cl(ψ)`. More precisely,

$$\text{Init}(\psi) := \{t \subseteq \text{cl}(\psi) \mid t \text{ is a type and } \Box\varphi \in t \text{ for each } \Box\varphi \in \text{cl}(\psi)\}.$$

- `Update(T) := $T \cup \text{supp}(T)$` , where `supp(T)` is the set of those types whose negated box formulas are witnessed by types in T . More precisely,

$$\text{supp}(T) := \{t \subseteq \text{cl}(\psi) \mid t \text{ is a type and,} \\ \text{for all } \neg\Box\varphi \in t, \text{ there exists } u \in T \\ \text{with } \neg\varphi \in u \text{ and } \Delta(t, u)\}.$$

Intuitively, the bottom-up algorithm starts with the set of types having no possibilities $\Diamond\varphi$, and adds those types whose possibilities are witnessed by a type in the set. Notice that the two algorithms described above, correspond to the two ways in which non-emptiness can be tested for automata for K .

32.5.2. Optimizations

The decision procedure described in the previous section handle the formula in three steps. First, the formula is converted into BNF. Then the initial set of

types is generated – we can think of this set as having some memory efficient representation. Finally, this set is updated through a fixpoint process. The answer of the decision procedure depends on a simple syntactic check of this fixpoint. In the following we consider three orthogonal optimizations techniques. See [PSV06] for more details, and for a description of preprocessing techniques that may further improve the performances of the OBDD-based implementations.

32.5.2.1. Particles

The approaches presented so far strongly depend on the fact that the BNF is used and they can be said to be redundant: if a type contains two conjuncts of some subformula of the input, then it also contains the corresponding conjunction – although the truth value of the latter is determined by the truth values of the former. Working with a different normal form it is possible to reduce such redundancy. We consider K -formulas in NNF (negation normal form) and we assume hereafter that all the formulas are in NNF. A set $p \subseteq \text{sub}(\psi)$ is a ψ -particle if it satisfies the following conditions:

- If $\varphi = \neg\varphi'$, then $\varphi \in p$ implies $\varphi' \notin p$
- If $\varphi = \varphi' \wedge \varphi''$, then $\varphi \in p$ implies $\varphi' \in p$ and $\varphi'' \in p$.
- If $\varphi = \varphi' \vee \varphi''$, then $\varphi \in p$ implies $\varphi' \in p$ or $\varphi'' \in p$.

Thus, in contrast to a type, a particle may contain both φ' and φ'' , but neither $\varphi' \wedge \varphi''$ nor $\varphi' \vee \varphi''$. Incidentally, particles are closer than types to assignments over modal atoms as described in Section 32.3.4. For particles, $\Delta(\cdot, \cdot)$ is defined as types. From a set of particles P and the corresponding $\Delta(\cdot, \cdot)$, a Kripke structure K_p can be constructed in the same way as from a set of types (see [PSV06]).

The schema presented in Figure 32.8 can be made to work for particles as well. In the top-down algorithm:

- $\text{Init}(\psi)$ is the set of all ψ -particles.
- $\text{Update}(P) := P \setminus \text{bad}(P)$, where $\text{bad}(P)$ is the particles in P that contain unwitnessed diamond formulas and it is defined similarly to the case of types

Also in the case of particles, the bottom-up approach differs only for the definitions of Init and Update :

- $\text{Init}(\psi) := \{p \subseteq \text{sub}(\psi) \mid p \text{ is a particle and } \diamond\varphi \notin p \text{ for all } \diamond\varphi \in \text{sub}(\psi)\}$ is the set of ψ -particles p that do not contain diamond formulas.
- $\text{Update}(P) := P \cup \text{supp}(P)$ where $\text{supp}(P)$ is the set of witnessed particles defined similarly to witnessed types.

Just like a set of types can be encoded in some efficient way, e.g., a set of bit vectors using a BDD, the same can be done for particles. It is easy to see that bit vectors for particles may be longer than bit vectors for types because, for example, the input may involve subformulas $\Box A$ and $\Diamond \neg A$. The overall size of the BDD may, however, be smaller for particles since particles impose fewer constraints than types, and improvements in the run time of the algorithms may result because the particle-based Update functions require checking less formulas than the type-based ones.

32.5.2.2. Lean approaches

Even though the particle approach imposes less constraints than the type approach, it still involves redundant information: like types, particles may contain both a conjunction and the corresponding conjuncts. To further reduce the size of the corresponding BDDs, in [PSV06] it is proposed a representation where “non-redundant” subformulas are only kept track of. A set of “non-redundant” subformulas $\mathbf{atom}(\psi)$ is defined as the set of those formulas in $\mathbf{cl}(\psi)$ that are neither conjunctions nor disjunctions, i.e., each $\varphi \in \mathbf{atom}(\psi)$ is of the form $\Box\varphi'$, A , $\neg\Box\varphi'$, or $\neg A$. By definition of types, each ψ -type $t \subseteq \mathbf{cl}(\psi)$, corresponds one-to-one to a *lean type* $\mathbf{lean}(t) := t \cap \mathbf{atom}(\psi)$. To specify algorithms for lean types, a relation $\dot{\in}$ must be defined recursively as follows: $\varphi \dot{\in} t$ if

- $\varphi \in \mathbf{atom}(\psi)$ and $\varphi \in t$,
- $\varphi = \neg\varphi'$ and not $\varphi' \dot{\in} t$,
- $\varphi = \varphi' \wedge \varphi''$, $\varphi' \dot{\in} t$, and $\varphi'' \dot{\in} t$, or
- $\varphi = \varphi' \vee \varphi''$, and $\varphi' \dot{\in} t$, or $\varphi'' \dot{\in} t$.

The top-down and bottom-up approach for types can be easily modified to work for lean types. It suffices to modify the definition of the functions **bad** and **supp** as follows:

$$\mathbf{bad}(T) := \{t \in T \mid \text{there exists } \neg\Box\varphi \in t \text{ and,} \\ \text{for all } u \in T \text{ with } \Delta(t, u), \text{ we have } \varphi \dot{\in} u\}.$$

$$\mathbf{supp}(T) := \{t \subseteq \mathbf{cl}(\psi) \mid t \text{ is a type and,} \\ \text{for all } \neg\Box\varphi \in t, \text{ there exists } u \in T \\ \text{with } \neg\varphi \dot{\in} u \text{ and } \Delta(t, u)\}.$$

A lean optimization can also be defined for particles – details are given in [PSV06]. Notice that this approach bears also some resemblances with the approach used in [CGH97] to translate LTL to SMV.

32.5.2.3. Level based evaluation

Another variation of the basic algorithm presented in Figure 32.8 exploits the fact that K enjoys the finite-tree-model property, i.e., each satisfiable formula ψ of K has a finite tree model of depth bounded by the depth of nested modal operators $\mathbf{depth}(\psi)$ of ψ . We can think of such a model as being partitioned into *layers*, where all states that are at distance i from the root are said to be in layer i . Instead of representing a complete model using a set of particles or types, each layer in the model can be represented using a separate set. Since only a subset of all subformulas appears in one layer, the representation can be more compact.

We start by (re)defining $\mathbf{cl}(\cdot)$ as

$$\mathbf{cl}_i(\psi) := \{\varphi \in \mathbf{cl}(\psi) \mid \varphi \text{ occurs at modal depth } i \text{ in } \psi\}$$

and $\Delta(\cdot, \cdot)$ as

$$\Delta(t, t') \text{ iff for all } t \subseteq \mathbf{cl}_i(\psi), t' \subseteq \mathbf{cl}_{i+1}(\psi), \text{ and } \varphi' \in t' \text{ for all } \Box\varphi' \in t.$$

```

d := depth( $\psi$ )
 $X_d := \text{Init}_d(\psi)$ 
for  $i := d - 1$  downto 0 do
   $X_i := \text{Update}(X_{i+1}, i)$ 
end
if exists  $x \in X_0$  such that  $\psi \in x$  then
  return " $\psi$  is satisfiable"
else
  return " $\psi$  is not satisfiable"

```

Figure 32.9. Algorithm for the level-based optimization.

in order to adapt them to the layered approach. A sequence of sets of types $T = \langle T_0, T_1, \dots, T_d \rangle$ with $T_i \subseteq 2^{\text{cl}_i(\psi)}$ can still be converted into a tree Kripke structure (see [PSV06] for details).

A bottom-up algorithm for level-based evaluation can be defined as in Figure 32.9. The algorithm works bottom-up in the sense that it starts with the leaves of a tree model *at the deepest level* and then moves up the tree model toward the root, adding nodes that are “witnessed”. In contrast, the bottom-up approach presented earlier starts with *all* leaves of a tree model. The accessory functions can be defined as follows:

- $\text{Init}_i(\psi) := \{t \subseteq \text{cl}_i(\psi) \mid t \text{ is a type}\}$.
- $\text{Update}(T, i) := \{t \in \text{Init}_i(\psi) \mid \text{for all } \neg \Box \varphi \in t \text{ there exists } u \in T \text{ with } \neg \varphi \in u \text{ and } \Delta_i(t, u)\}$.

For a set T of types of formulas at level $i + 1$, $\text{Update}(T, i)$ represents all types of formulas at level i that are witnessed in T .

32.6. The Eager DPLL-based approach

Recently [SV06, SV08] have explored the idea of encoding K_m/\mathcal{ALC} -satisfiability into SAT and handle it by state-of-the-art SAT tools. A satisfiability-preserving encoding from K_m/\mathcal{ALC} to SAT was proposed there, with a few variations and some important optimizations. As K_m -satisfiability is PSPACE-complete, the encoding is necessarily worst-case exponential (unless PSPACE=NP). However, the only source of exponentiality is the modal depth of the input formula: if the depth is bounded, the problem is NP-complete [Hal95], so that the encoding reduces to polynomial. In practice, the experiments presented there showed that this approach can handle most or all the problems which are at the reach of the other approaches, with performances which are comparable with, or even better than, those of the current state-of-the-art tools.

As this idea was inspired by the so-called “eager approach” to SMT [BGV99, SSB02, Str02] (see §26.3), we call this approach, *eager approach to modal reasoning*. In this section we present an overview of this approach.

32.6.1. The basic encoding

In order to make our presentation more uniform, and to avoid considering the polarity of subformulas, we adopt from [Fit83, Mas00] the representation of K_m -formulas from the following table:

α	α_1	α_2	β	β_1	β_2	π^r	π_0^r	ν^r	ν_0^r
$(\varphi_1 \wedge \varphi_2)$	φ_1	φ_2	$(\varphi_1 \vee \varphi_2)$	φ_1	φ_2	$\diamond_r \varphi_1$	φ_1	$\square_r \varphi_1$	φ_1
$\neg(\varphi_1 \vee \varphi_2)$	$\neg\varphi_1$	$\neg\varphi_2$	$\neg(\varphi_1 \wedge \varphi_2)$	$\neg\varphi_1$	$\neg\varphi_2$	$\neg\square_r \varphi_1$	$\neg\varphi_1$	$\neg\diamond_r \varphi_1$	$\neg\varphi_1$
$\neg(\varphi_1 \rightarrow \varphi_2)$	φ_1	$\neg\varphi_2$	$(\varphi_1 \rightarrow \varphi_2)$	$\neg\varphi_1$	φ_2				

in which non-literal K_m -formulas are grouped into four categories: α 's (conjunctive), β 's (disjunctive), π 's (existential), ν 's (universal). All such formulas occur in the main formula with positive polarity only.¹⁴ This allows for disregarding the issue of polarity of subformulas.

We borrow some notation from the *Single Step Tableau (SST)* framework [Mas00, DM00]. We represent univocally states in \mathcal{M} as labels σ , represented as non empty sequences of integers $1.n_1^{r_1}.n_2^{r_2} \dots .n_k^{r_k}$, s.t. the label 1 represents the root state, and $\sigma.n^r$ represents the n -th successor of σ through the relation \mathcal{R}_r . With a little abuse of notation, hereafter we may say “a state σ ” meaning “a state labeled by σ ”. We call a *labeled formula* a pair $\langle \sigma : \psi \rangle$, s.t. σ is a state label and ψ is a K_m -formula.

Let $A_{[\]}$ be an *injective* function which maps a labeled formula $\langle \sigma : \psi \rangle$, s.t. is not in the form $\neg\phi$, into a Boolean variable $A_{[\sigma, \psi]}$. Let $L_{[\sigma, \psi]}$ denote $\neg A_{[\sigma, \phi]}$ if ψ is in the form $\neg\phi$, $A_{[\sigma, \psi]}$ otherwise. Given a K_m -formula φ , K_m2SAT builds a Boolean CNF formula recursively as follows:

$$K_m2SAT(\varphi) := A_{[1, \varphi]} \wedge Def(1, \varphi) \quad (32.21)$$

$$Def(\sigma, A_i) := \top \quad (32.22)$$

$$Def(\sigma, \neg A_i) := \top \quad (32.23)$$

$$Def(\sigma, \alpha) := (L_{[\sigma, \alpha]} \rightarrow (L_{[\sigma, \alpha_1]} \wedge L_{[\sigma, \alpha_2]})) \wedge Def(\sigma, \alpha_1) \wedge Def(\sigma, \alpha_2) \quad (32.24)$$

$$Def(\sigma, \beta) := (L_{[\sigma, \beta]} \rightarrow (L_{[\sigma, \beta_1]} \vee L_{[\sigma, \beta_2]})) \wedge Def(\sigma, \beta_1) \wedge Def(\sigma, \beta_2) \quad (32.25)$$

$$Def(\sigma, \pi^{r,j}) := (L_{[\sigma, \pi^{r,j}]} \rightarrow L_{[\sigma,j, \pi_0^{r,j}]}) \wedge Def(\sigma,j, \pi_0^{r,j}) \quad (32.26)$$

$$Def(\sigma, \nu^r) := \bigwedge_{\langle \sigma : \pi^{r,i} \rangle} ((L_{[\sigma, \nu^r]} \wedge L_{[\sigma, \pi^{r,i}]} \rightarrow L_{[\sigma,i, \nu_0^r]}) \wedge \bigwedge_{\langle \sigma : \pi^{r,i} \rangle} Def(\sigma,i, \nu_0^r)). \quad (32.27)$$

Here by “ $\langle \sigma : \pi^{r,i} \rangle$ ” we mean that $\pi^{r,i}$ is the j -th distinct π^r formula labeled by σ .

We assume that the K_m -formulas are represented as DAGs, so to avoid the expansion of the same $Def(\sigma, \psi)$ more than once. Moreover, following [Mas00], we assume that, for each σ , the $Def(\sigma, \psi)$'s are expanded in the order: α, β, π, ν . Thus, each $Def(\sigma, \nu^r)$ is expanded after the expansion of all $Def(\sigma, \pi^{r,i})$'s, so that $Def(\sigma, \nu^r)$ will generate one clause $((L_{[\sigma, \pi^{r,i}]} \wedge L_{[\sigma, \pi^{r,i}]} \rightarrow L_{[\sigma,i, \nu_0^r]})$ and one novel definition $Def(\sigma,i, \nu_0^r)$ for each $Def(\sigma, \pi^{r,i})$ expanded.¹⁵

Intuitively, $K_m2SAT(\varphi)$ mimics the construction of an SST tableau expansion [Mas00, DM00], s.t., if there exists an open tableau \mathcal{T} for $\langle 1 : \varphi \rangle$, then

¹⁴E.g., a \wedge -formula [resp. \vee -formula] occurring negatively is considered a positive occurrence of a β -formula [resp. an α -formula]; a \square_r -formula [resp. a \diamond_r -formula] occurring negatively is considered a positive occurrence of a π -formula [resp. a ν -formula].

¹⁵Notice that, e.g., an occurrence of $\square_r \psi$ is considered a ν -formula if positive, a π -formula if negative.

there exists a total truth assignment μ which satisfies $K_m2SAT(\varphi)$, and vice versa. Thus, from the correctness and completeness of the SST framework, we have the following fact.

Theorem 2. [SV08] *A K_m -formula φ is K_m -satisfiable if and only if the corresponding Boolean formula $K_m2SAT(\varphi)$ is satisfiable.*

Notice that, due to (32.27), the number of variables and clauses in $K_m2SAT(\varphi)$ may grow exponentially with $depth(\varphi)$. This is in accordance to what stated in [Hal95].

Example 9 (NNF). *Let φ_{nnf} be $(\diamond A_1 \vee \diamond(A_2 \vee A_3)) \wedge \square \neg A_1 \wedge \square \neg A_2 \wedge \square \neg A_3$.¹⁶ It is easy to see that φ_{nnf} is K_1 -unsatisfiable: the \diamond -atoms impose that at least one atom A_i is true in at least one successor of the root state, whilst the \square -atoms impose that all atoms A_i are false in all successor states of the root state. $K_m2SAT(\varphi_{nnf})$ is:*

1. $A_{[1, \varphi_{nnf}]}$
2. $\wedge (A_{[1, \varphi_{nnf}]} \rightarrow (A_{[1, \diamond A_1 \vee \diamond(A_2 \vee A_3)]} \wedge A_{[1, \square \neg A_1]} \wedge A_{[1, \square \neg A_2]} \wedge A_{[1, \square \neg A_3]}))$
3. $\wedge (A_{[1, \diamond A_1 \vee \diamond(A_2 \vee A_3)]} \rightarrow (A_{[1, \diamond A_1]} \vee A_{[1, \diamond(A_2 \vee A_3)]}))$
4. $\wedge (A_{[1, \diamond A_1]} \rightarrow A_{[1.1, A_1]})$
5. $\wedge (A_{[1, \diamond(A_2 \vee A_3)]} \rightarrow A_{[1.2, A_2 \vee A_3]})$
6. $\wedge ((A_{[1, \square \neg A_1]} \wedge A_{[1, \diamond A_1]}) \rightarrow \neg A_{[1.1, A_1]})$
7. $\wedge ((A_{[1, \square \neg A_2]} \wedge A_{[1, \diamond A_1]}) \rightarrow \neg A_{[1.1, A_2]})$
8. $\wedge ((A_{[1, \square \neg A_3]} \wedge A_{[1, \diamond A_1]}) \rightarrow \neg A_{[1.1, A_3]})$
9. $\wedge ((A_{[1, \square \neg A_1]} \wedge A_{[1, \diamond(A_2 \vee A_3)]}) \rightarrow \neg A_{[1.2, A_1]})$
10. $\wedge ((A_{[1, \square \neg A_2]} \wedge A_{[1, \diamond(A_2 \vee A_3)]}) \rightarrow \neg A_{[1.2, A_2]})$
11. $\wedge ((A_{[1, \square \neg A_3]} \wedge A_{[1, \diamond(A_2 \vee A_3)]}) \rightarrow \neg A_{[1.2, A_3]})$
12. $\wedge (A_{[1.2, A_2 \vee A_3]} \rightarrow (A_{[1.2, A_2]} \vee A_{[1.2, A_3]}))$

After a run of BCP, 3. reduces to the implicate disjunction $A_{[1, \diamond A_1]} \vee A_{[1, \diamond(A_2 \vee A_3)]}$. If the first element $A_{[1, \diamond A_1]}$ is assigned to true, then by BCP we have a conflict on 4. and 6. If $A_{[1, \diamond A_1]}$ is set to false, then the second element $A_{[1, \diamond(A_2 \vee A_3)]}$ is assigned to true, and by BCP we have a conflict on 12. Thus $K_m2SAT(\varphi_{nnf})$ is unsatisfiable.

32.6.2. Optimizations

The following optimizations of the encoding have been proposed in [SV06, SV08] in order to reduce the size of the output propositional formula.

32.6.2.1. Pre-conversion to BNF

Before the encoding, some potentially useful preprocessing on the input formula can be performed. First, the input K_m -formulas can be converted into BNF. One potential advantage is that, when one $\square_r \psi$ occurs both positively and negatively (like, e.g., in $(\square_r \psi \vee \dots) \wedge (\neg \square_r \psi \vee \dots) \wedge \dots$), then both occurrences of $\square_r \psi$ are labeled by the same Boolean atom $A_{[\sigma, \square_r \psi]}$, and hence they are always assigned the

¹⁶For K_1 -formulas, we omit the box and diamond indexes.

same truth value by DPLL; with NNF, instead, the negative occurrence $\neg\Box_r\psi$ is rewritten into $\Diamond_r\neg\psi$, so that two distinct Boolean atoms $A_{[\sigma, \Box_r\psi]}$ and $A_{[\sigma, \Diamond_r\neg\psi]}$ are generated; DPLL can assign them the same truth value, creating a hidden conflict which may require some extra Boolean search to reveal.

Example 10 (BNF). *We consider the BNF variant of the φ_{nnf} formula of Example 9, $\varphi_{bnf} = (\neg\Box\neg A_1 \vee \neg\Box(\neg A_2 \wedge \neg A_3)) \wedge \Box\neg A_1 \wedge \Box\neg A_2 \wedge \Box\neg A_3$. As before, it is easy to see that φ_{bnf} is K_1 -unsatisfiable. $K_m2SAT(\varphi_{bnf})$ is:*

1. $A_{[1, \varphi_{bnf}]}$
2. $\wedge (A_{[1, \varphi_{bnf}]} \rightarrow (A_{[1, (\neg\Box\neg A_1 \vee \neg\Box(\neg A_2 \wedge \neg A_3))]} \wedge A_{[1, \Box\neg A_1]} \wedge A_{[1, \Box\neg A_2]} \wedge A_{[1, \Box\neg A_3]}))$
3. $\wedge (A_{[1, (\neg\Box\neg A_1 \vee \neg\Box(\neg A_2 \wedge \neg A_3))]} \rightarrow (\neg A_{[1, \Box\neg A_1]} \vee \neg A_{[1, \Box(\neg A_2 \wedge \neg A_3)]}))$
4. $\wedge (\neg A_{[1, \Box\neg A_1]} \rightarrow A_{[1.1, A_1]})$
5. $\wedge (\neg A_{[1, \Box(\neg A_2 \wedge \neg A_3)]} \rightarrow \neg A_{[1.2, (\neg A_2 \wedge \neg A_3)]})$
6. $\wedge ((A_{[1, \Box\neg A_1]} \wedge \neg A_{[1, \Box\neg A_1]}) \rightarrow \neg A_{[1.1, A_1]})$
7. $\wedge ((A_{[1, \Box\neg A_2]} \wedge \neg A_{[1, \Box\neg A_1]}) \rightarrow \neg A_{[1.1, A_2]})$
8. $\wedge ((A_{[1, \Box\neg A_3]} \wedge \neg A_{[1, \Box\neg A_1]}) \rightarrow \neg A_{[1.1, A_3]})$
9. $\wedge ((A_{[1, \Box\neg A_1]} \wedge \neg A_{[1, \Box(\neg A_2 \wedge \neg A_3)]}) \rightarrow \neg A_{[1.2, A_1]})$
10. $\wedge ((A_{[1, \Box\neg A_2]} \wedge \neg A_{[1, \Box(\neg A_2 \wedge \neg A_3)]}) \rightarrow \neg A_{[1.2, A_2]})$
11. $\wedge ((A_{[1, \Box\neg A_3]} \wedge \neg A_{[1, \Box(\neg A_2 \wedge \neg A_3)]}) \rightarrow \neg A_{[1.2, A_3]})$
12. $\wedge (\neg A_{[1.2, (\neg A_2 \wedge \neg A_3)]} \rightarrow (A_{[1.2, A_2]} \vee A_{[1.2, A_3]}))$

Unlike with NNF, $K_m2SAT(\varphi_{bnf})$ is found unsatisfiable directly by BCP. In fact, the unit-propagation of $A_{[1, \Box\neg A_1]}$ from 2. causes $\neg A_{[1, \Box\neg A_1]}$ in 3. to be false, so that one of the two (unsatisfiable) branches induced by the disjunction is cut a priori. With NNF, the corresponding atoms $A_{[1, \Box\neg A_1]}$ and $A_{[1, \Diamond A_1]}$ are not recognized to be one the negation of the other, s.t. DPLL may need exploring one Boolean branch more.

32.6.2.2. Lifting boxes and diamonds

The second form of preprocessing is, the K_m -formula can also be rewritten by recursively applying the K_m -validity-preserving “box/diamond-lifting rules”:

$$\begin{aligned} (\Box_r\varphi_1 \wedge \Box_r\varphi_2) &\Longrightarrow \Box_r(\varphi_1 \wedge \varphi_2), & (\Diamond_r\varphi_1 \vee \Diamond_r\varphi_2) &\Longrightarrow \Diamond_r(\varphi_1 \vee \varphi_2), \\ (\neg\Box_r\varphi_1 \vee \neg\Box_r\varphi_2) &\Longrightarrow \neg\Box_r(\varphi_1 \wedge \varphi_2), & (\neg\Diamond_r\varphi_1 \wedge \neg\Diamond_r\varphi_2) &\Longrightarrow \neg\Diamond_r(\varphi_1 \vee \varphi_2). \end{aligned} \quad (32.28)$$

This has the potential benefit of reducing the number of $\pi^{r,i}$ formulas, and hence the number of labels $\sigma.i$ to take into account in the expansion of the $Def(\sigma, \nu^r)$'s (32.27).

Example 11 (BNF with LIFT). *If we apply the rules (32.28) to the formula of Example 10, then we have $\varphi_{bnflift} = \neg\Box(\neg A_1 \wedge \neg A_2 \wedge \neg A_3) \wedge \Box(\neg A_1 \wedge \neg A_2 \wedge \neg A_3)$. $K_m2SAT(\varphi_{bnflift})$ is thus:*

1. $A_{[1, \varphi_{bnflift}]}$
2. $\wedge (A_{[1, \varphi_{bnflift}]} \rightarrow (\neg A_{[1, \Box(\neg A_1 \wedge \neg A_2 \wedge \neg A_3)]} \wedge A_{[1, \Box(\neg A_1 \wedge \neg A_2 \wedge \neg A_3)]}))$
3. $\wedge (\neg A_{[1, \Box(\neg A_1 \wedge \neg A_2 \wedge \neg A_3)]} \rightarrow \neg A_{[1.1, (\neg A_1 \wedge \neg A_2 \wedge \neg A_3)]})$
4. $\wedge ((A_{[1, \Box(\neg A_1 \wedge \neg A_2 \wedge \neg A_3)]} \wedge \neg A_{[1, \Box(\neg A_1 \wedge \neg A_2 \wedge \neg A_3)]}) \rightarrow A_{[1.1, (\neg A_1 \wedge \neg A_2 \wedge \neg A_3)]})$
5. $\wedge (\neg A_{[1.1, (\neg A_1 \wedge \neg A_2 \wedge \neg A_3)]} \rightarrow (A_{[1.1, A_1]} \vee A_{[1.1, A_2]} \vee A_{[1.1, A_3]}))$
6. $\wedge (A_{[1.1, (\neg A_1 \wedge \neg A_2 \wedge \neg A_3)]} \rightarrow (\neg A_{[1.1, A_1]} \wedge \neg A_{[1.1, A_2]} \wedge \neg A_{[1.1, A_3]}))$

$K_m2SAT(\varphi_{bnflift})$ is found unsatisfiable directly by BCP on 1. and 2..

One potential drawback of applying the lifting rules (32.28) is that, by collapsing a conjunction/disjunction of modal atoms into one single atom, the possibility of sharing box/diamond subformulas in the DAG representation of φ is reduced. To cope with this problem, it is possible to adopt a *controlled* policy for applying Box/Diamond-lifting, that is, to apply (32.28) only if neither atom has multiple occurrences.

32.6.2.3. Handling incompatible π^r and ν^r

A first straightforward optimization, in the BNF variant, avoids the useless encoding of incompatible π^r and ν^r formulas. In BNF, in fact, the same subformula $\Box_r\psi$ may occur in the same state σ both positively and negatively (e.g., if $\pi^{r:j}$ is $\neg\Box_r\psi$ and ν^r is $\Box_r\psi$). If so, K_m2SAT labels both those occurrences of $\Box_r\psi$ with the same Boolean atom $A_{[\sigma, \Box_r\psi]}$, and produces recursively two distinct subsets of clauses in the encoding, by applying (32.26) to $\neg\Box_r\psi$ and (32.27) to $\Box_r\psi$ respectively. However, the latter step (32.27) generates a *valid* clause $(A_{[\sigma, \Box_r\psi]} \wedge \neg A_{[\sigma, \Box_r\psi]}) \rightarrow A_{[\sigma.j, \psi]}$, which can be dropped. Consequently $A_{[\sigma.j, \psi]}$ no more occurs in the formula, so that also $Def(\sigma.i, \psi)$ can be dropped as well, as there is no more need of defining $\langle\sigma : \psi\rangle$.

Example 12. In the formula φ_{bnf} of Example 10 the implication 6. is valid and can be dropped. In the formula $\varphi_{bnflift}$ of Example 11, not only 4., but also 6. can be dropped.

32.6.2.4. On-the-fly Boolean Constraint Propagation

One major problem of the basic encoding of §32.6.1 is that it is *purely-syntactic*, that is, it does not consider the possible truth values of the subformulas, and the effect of their propagation through the Boolean and modal connectives. In particular, K_m2SAT applies (32.26) [resp. (32.27)] to *every* π -subformula [resp. ν -subformula], regardless the fact that the truth values which can be deterministically assigned to the labeled subformulas of $\langle 1 : \varphi \rangle$ may allow for dropping some labeled π -/ ν -subformulas, and thus prevent the need of encoding them.

One solution to this problem is that of applying BCP on-the-fly during the construction of $K_m2SAT(\varphi)$. If a contradiction is found, then $K_m2SAT(\varphi)$ is \perp . When BCP allows for dropping one implication in (32.24)-(32.27) without assigning some of its implicate literals, namely $L_{[\sigma, \psi_i]}$, then $\langle\sigma : \psi_i\rangle$ needs not to be defined, so that $Def(\sigma, \psi)$ can be dropped. Importantly, dropping $Def(\sigma, \pi^{r:j})$ for some π -formula $\langle\sigma : \pi^{r:j}\rangle$ prevents generating the label $\sigma.j$ (32.26) and all its successor labels $\sigma.j.\sigma'$ (corresponding to the subtree of states rooted in $\sigma.j$), so that all the corresponding labeled subformulas are not encoded.

Example 13. Consider Example 10. After building 1. – 3. in $K_m2SAT(\varphi_{bnf})$, the atoms $A_{[1, \varphi_{bnf}]}$, $A_{[1, (\neg\Box\neg A_1 \vee \neg\Box(\neg A_2 \wedge \neg A_3))]}$, $A_{[1, \Box\neg A_1]}$, $A_{[1, \Box\neg A_2]}$ and $A_{[1, \Box\neg A_3]}$ can be deterministically assigned to true by applying BCP. This causes the removal from 3. of the first-implicated disjunct $\neg A_{[1, \Box\neg A_1]}$, so that 4. is not generated. As label 1.1. is not defined, 6., 7. and 8. are not generated. Then after the construction of 5., 9., 10., 11. and 12., by applying BCP a contradiction is found, so that $K_m2SAT(\varphi)$ is \perp .

32.6.2.5. On-the-fly Pure-Literal Reduction

Another technique, evolved from that proposed in [PSV02, PV03], applies Pure-Literal reduction on-the-fly during the construction of $K_m2SAT(\varphi)$. When for some label σ all the clauses containing atoms $A_{[\sigma, \psi]}$ have been generated, if some of them occurs only positively [resp. negatively], then it can be safely assigned to true [resp. to false], and hence the clauses containing $A_{[\sigma, \psi]}$ can be dropped. As a consequence, some other atom $A_{[\sigma, \psi']}$ can become pure, so that the process is repeated until a fixpoint is reached.

Example 14. Consider the formula φ_{bnf} of Example 10. During the construction of $K_m2SAT(\varphi_{bnf})$, after 1.-8. are generated, no more clause containing atoms in the form $A_{[1.1, \psi]}$ is to be generated. Then we notice that $A_{[1.1, A_2]}$ and $A_{[1.1, A_3]}$ occur only negatively, so that they can be safely assigned to false. Therefore, 7. and 8. can be safely dropped. Same discourse applies lately to $A_{[1.2, A_1]}$ and 9. The resulting formula is found inconsistent by BCP. (In fact, notice that in Example 10 $A_{[1.1, A_2]}$, $A_{[1.1, A_3]}$, and $A_{[1.2, A_1]}$ play no role in the unsatisfiability of $K_m2SAT(\varphi_{bnf})$.)

References

- [ABC⁺02] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions. In *Proceedings of 18th International Conference on Automated Deduction (CADE)*, volume 2392 of *LNAI*. Springer, 2002.
- [ACG00] A. Armando, C. Castellini, and E. Giunchiglia. SAT-based procedures for temporal reasoning. In *Proceedings of 5th European Conference on Planning, (ECP)*, volume 1809 of *LNCIS*. Springer, 2000.
- [AG93] A. Armando and E. Giunchiglia. Embedding Complex Decision Procedures inside an Interactive Theorem Prover. *Annals of Mathematics and Artificial Intelligence*, 8(3–4):475–502, 1993.
- [AGHd00] C. Areces, R. Gennari, J. Heguiabehere, and M. de Rijke. Tree-based heuristics in modal theorem proving. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI)*, pages 199–203, 2000.
- [BCM⁺03] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.
- [BFH⁺94] F. Baader, E. Franconi, B. Hollunder, B. Nebel, and H. J. Profitlich. An Empirical Analysis of Optimization Techniques for Terminological Representation Systems or: Making KRIS get a move on. *Applied Artificial Intelligence. Special Issue on Knowledge Base Management*, 4:109–132, 1994.
- [BFT95] P. Bresciani, E. Franconi, and S. Tessaris. Implementing and testing expressive Description Logics: a preliminary report. In *Proc. International Workshop on Description Logics*, Rome, Italy, 1995.

- [BGdR03] S. Brand, R. Gennari, and M. de Rijke. Constraint Programming for Modelling and Solving Modal Satisfiability. In *Proceedings of 9th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 3010 of *LNAI*, pages 795–800. Springer, 2003.
- [BGV99] R. Bryant, S. German, and M. Velev. Exploiting Positive Equality in a Logic of Equality with Uninterpreted Functions. In *Proceedings of 11th International Conference on Computer Aided Verification (CAV)*, volume 1633 of *LNCS*. Springer, 1999.
- [BH91] F. Baader and B. Hollunder. A Terminological Knowledge Representation System with Complete Inference Algorithms. In *Proceedings of the First International Workshop on Processing Declarative Knowledge*, volume 572 of *LNCS*, pages 67–85, Kaiserslautern (Germany), 1991. Springer-Verlag.
- [Bra01] R. Brafman. A simplifier for propositional formulas with many binary clauses. In *Proceedings of 17th International Joint Conference on Artificial Intelligence (IJCAI)*, 2001.
- [BS97] R. J. Bayardo and R. C. Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT instances. In *Proceedings of 14th National Conference on Artificial Intelligence (AAAI)*, pages 203–208. AAAI Press, 1997.
- [BW03] F. Bacchus and J. Winter. Effective Preprocessing with Hyper-Resolution and Equality Reduction. In *Proceedings of 6th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2003.
- [CGH97] E. M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at ltl model checking. *Formal Methods in System Design*, 10(1):47–71, 1997.
- [Che80] B. F. Chellas. *Modal Logic – an Introduction*. Cambridge University Press, 1980.
- [D’A92] M. D’Agostino. Are Tableaux an Improvement on Truth-Tables? *Journal of Logic, Language and Information*, 1:235–252, 1992.
- [DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Journal of the ACM*, 5(7), 1962.
- [DM94] M. D’Agostino and M. Mondadori. The Taming of the Cut. *Journal of Logic and Computation*, 4(3):285–319, 1994.
- [DM00] F. Donini and F. Massacci. EXPTIME tableaux for ALC. *Artificial Intelligence*, 124(1):87–138, 2000.
- [DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.
- [EB05] N. Eén and A. Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. In *Proceedings of 8th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 3569 of *LNCS*. Springer, 2005.
- [ES04] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proceedings of 6th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 2919 of *LNCS*, pages 502–518. Springer,

2004.

- [FHMV95] R. Fagin, J. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about knowledge*. The MIT press, 1995.
- [Fit83] M. Fitting. *Proof Methods for Modal and Intuitionistic Logics*. D. Reidel Publishg, 1983.
- [GGST98] E. Giunchiglia, F. Giunchiglia, R. Sebastiani, and A. Tacchella. More evaluation of decision procedures for modal logics. In *Proceedings of Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, Trento, Italy, 1998.
- [GGST00] E. Giunchiglia, F. Giunchiglia, R. Sebastiani, and A. Tacchella. SAT vs. Translation based decision procedures for modal logics: a comparative evaluation. *Journal of Applied Non-Classical Logics*, 10(2):145–172, 2000.
- [GGT01] E. Giunchiglia, F. Giunchiglia, and A. Tacchella. SAT Based Decision Procedures for Classical Modal Logics. *Journal of Automated Reasoning*. Special Issue: Satisfiability at the start of the year 2000, 2001.
- [GN02] E. Goldberg and Y. Novikov. BerkMin: A Fast and Robust SAT-Solver. In *Proc. DATE '02*, page 142, Washington, DC, USA, 2002. IEEE Computer Society.
- [GRS96] F. Giunchiglia, M. Roveri, and R. Sebastiani. A new method for testing decision procedures in modal and terminological logics. In *Proc. of 1996 International Workshop on Description Logics - DL'96*, Cambridge, MA, USA, November 1996.
- [GS96a] F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures - the case study of modal K. In *Proc. CADE'13*, LNAI, New Brunswick, NJ, USA, August 1996. Springer.
- [GS96b] F. Giunchiglia and R. Sebastiani. A SAT-based decision procedure for ALC. In *Proc. of the 5th International Conference on Principles of Knowledge Representation and Reasoning - KR'96*, Cambridge, MA, USA, November 1996.
- [GS00] F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures - the case study of modal K(m). *Information and Computation*, 162(1/2), October/November 2000.
- [GSK98] C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*, pages 431–437, Madison, Wisconsin, 1998.
- [GT01] E. Giunchiglia and A. Tacchella. Testing for Satisfiability in Modal Logics using a Subset-matching Size-bounded cache. *Annals of Mathematics and Artificial Intelligence*, 33:39–68, 2001.
- [Hal95] J. Y. Halpern. The effect of bounding the number of primitive propositions and the depth of nesting on the complexity of modal logic. *Artificial Intelligence*, 75(3):361–372, 1995.
- [HJSS96] A. Heuerding, G. Jager, S. Schwendimann, and M. Seyfried. The

- Logics Workbench LWB: A Snapshot. *Euromath Bulletin*, 2(1):177–186, 1996.
- [HM85] J. Y. Halpern and Y. Moses. A guide to the modal logics of knowledge and belief: preliminary draft. In *Proceedings of 9th International Joint Conference on Artificial Intelligence*, pages 480–490, Los Angeles, CA, 1985. Morgan Kaufmann Publ. Inc.
- [HM92] J. Y. Halpern and Y. Moses. A guide to the completeness and complexity for modal logics of knowledge and belief. *Artificial Intelligence*, 54(3):319–379, 1992.
- [HM01] V. Haarslev and R. Moeller. RACER System Description. In *Proc. of International Joint Conference on Automated Reasoning - IJCAR-2001*, volume 2083 of *LNAI*, Siena, Italy, July 2001. Springer-verlag.
- [Hor98a] I. Horrocks. The FaCT system. In *Proc. Automated Reasoning with Analytic Tableaux and Related Methods: International Conference Tableaux'98*, number 1397 in *LNAI*, pages 307–312. Springer, May 1998.
- [Hor98b] I. Horrocks. Using an expressive description logic: FaCT or fiction? In *Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 636–647, 1998.
- [HPS99] I. Horrocks and P. F. Patel-Schneider. Optimizing Description Logic Subsumption. *Journal of Logic and Computation*, 9(3):267–293, 1999.
- [HPSS00] I. Horrocks, P. F. Patel-Schneider, and R. Sebastiani. An Analysis of Empirical Testing for Modal Decision Procedures. *Logic Journal of the IGPL*, 8(3):293–323, May 2000.
- [HS96] A. Heurding and S. Schwendimann. A benchmark method for the propositional modal logics K, KT, S4. Technical Report IAM-96-015, University of Bern, Switzerland, 1996.
- [HS99] U. Hustadt and R. Schmidt. An empirical analysis of modal theorem provers. *Journal of Applied Non-Classical Logics*, 9(4), 1999.
- [HSW99] U. Hustadt, R. A. Schmidt, and C. Weidenbach. MSPASS: Subsumption Testing with SPASS. In *Proc. 1999 International Workshop on Description Logics (DL'99)*, vol. 22, *CEUR Workshop Proceedings*, pages 136–137, 1999.
- [Lad77] R. Ladner. The computational complexity of provability in systems of modal propositional logic. *SIAM J. Comp.*, 6(3):467–480, 1977.
- [Mas94] F. Massacci. Strongly analytic tableaux for normal modal logics. In *In Proceedings of 12th International Conference on Automated Deduction*, volume 814 of *Lecture Notes in Computer Science*. Springer, 1994.
- [Mas98] F. Massacci. Simplification: A general constraint propagation technique for propositional and modal tableaux. In *Proc. 2nd International Conference on Analytic Tableaux and Related Methods (TABLEAUX-97)*, volume 1397 of *LNAI*. Springer, 1998.
- [Mas99] F. Massacci. Design and Results of Tableaux-99 Non-Classical (Modal) System Competition. In *Automated Reasoning with Analytic Tableaux and Related Methods: International Conference (Tableaux'99)*, 1999.

- [Mas00] F. Massacci. Single Step Tableaux for modal logics: methodology, computations, algorithms. *Journal of Automated Reasoning*, Vol. 24(3), 2000.
- [MMZ⁺01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, 2001.
- [MSS96] J. P. Marques-Silva and K. A. Sakallah. GRASP - A new Search Algorithm for Satisfiability. In *Proc. ICCAD'96*, 1996.
- [Ngu05] L. A. Nguyen. On the Complexity of Fragments of Modal Logics. In *Advances in Modal Logic*. King's College Publications, 2005.
- [NOT06] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, November 2006.
- [PS98] P. F. Patel-Schneider. DLP system description. In *Proc. DL-98*, pages 87–89, 1998.
- [PSS01] P. F. Patel-Schneider and R. Sebastiani. A system and methodology for generating random modal formulae. In *Proc. IJCAR-2001*, volume 2083 of *LNAI*. Springer-verlag, 2001.
- [PSS03] P. F. Patel-Schneider and R. Sebastiani. A New General Method to Generate Random Modal Formulae for Testing Decision Procedures. *Journal of Artificial Intelligence Research, (JAIR)*, 18:351–389, May 2003. Morgan Kaufmann.
- [PSV02] G. Pan, U. Sattler, and M. Y. Vardi. BDD-Based Decision Procedures for K. In *In proceedings of 18th International Conference on Automated Deduction*, volume 2392 of *Lecture Notes in Computer Science*. Springer, 2002.
- [PSV06] G. Pan, U. Sattler, and M. Y. Vardi. BDD-Based Decision Procedures for the Modal Logic K. *Journal of Applied Non-Classical Logics*, 16(2), 2006.
- [PV03] G. Pan and M. Y. Vardi. Optimizing a BDD-based modal solver. In *Proceedings of 19th International Conference on Automated Deduction*, volume 2741 of *Lecture Notes in Computer Science*. Springer, 2003.
- [Sch91] K. D. Schild. A correspondence theory for terminological logics: preliminary report. In *Proc. 12th Int. Joint Conf. on Artificial Intelligence, IJCAI*, Sydney, Australia, 1991.
- [Seb01] R. Sebastiani. Integrating SAT Solvers with Math Reasoners: Foundations and Basic Algorithms. Technical Report 0111-22, ITC-IRST, Trento, Italy, November 2001.
- [Seb07] R. Sebastiani. Lazy Satisfiability Modulo Theories. *Journal on Satisfiability, Boolean Modeling and Computation – JSAT.*, 3, 2007.
- [Smu68] R. M. Smullyan. *First-Order Logic*. Springer-Verlag, NY, 1968.
- [SSB02] O. Strichman, S. Seshia, and R. Bryant. Deciding separation formulas with SAT. In *Proc. of Computer Aided Verification, (CAV'02)*, LNCS. Springer, 2002.
- [SSS91] M. Schmidt-Schauß and G. Smolka. Attributive Concept Descriptions

- with Complements. *Artificial Intelligence*, 48:1–26, 1991.
- [Str02] O. Strichman. On Solving Presburger and Linear Arithmetic with SAT. In *Proc. of Formal Methods in Computer-Aided Design (FMCAD 2002)*, LNCS. Springer, 2002.
- [SV98] R. Sebastiani and A. Villafiorita. SAT-based decision procedures for normal modal logics: a theoretical framework. In *Proc. AIMS'98*, volume 1480 of *LNAI*. Springer, 1998.
- [SV06] R. Sebastiani and M. Vescovi. Encoding the Satisfiability of Modal and Description Logics into SAT: The Case Study of $K(m)/ALC$. In *Proc. SAT'06*, volume 4121 of *LNCS*. Springer, 2006.
- [SV08] R. Sebastiani and M. Vescovi. Automated Reasoning in Modal and Description Logics via SAT Encoding: the Case Study of $K(m)/ALC$ -Satisfiability. Technical report, DISI, University of Trento, Italy, 2008. Submitted for journal publication. Available as <http://disi.unitn.it/~rseba/sat06/>.
- [Tac99] A. Tacchella. *SAT system description. In *Proc. 1999 International Workshop on Description Logics (DL'99)*, vol. 22, *CEUR Workshop Proceedings*, pages 142–144, 1999.
- [Tin02] C. Tinelli. A DPLL-based Calculus for Ground Satisfiability Modulo Theories. In *Proc. JELIA-02*, volume 2424 of *LNAI*, pages 308–319. Springer, 2002.
- [Var89] M. Y. Vardi. On the complexity of epistemic reasoning. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, pages 243–252, 1989.
- [Vor99] A. Voronkov. KK: a theorem prover for K. In *CADE-16: Proceedings of the 16th International Conference on Automated Deduction*, number 1632 in *LNAI*, pages 383–387. Springer, 1999.
- [Vor01] A. Voronkov. How to optimize proof-search in modal logics: new methods of proving redundancy criteria for sequent calculi. *ACM Transactions on Computational Logic*, 2(2):182–215, 2001.
- [WW99] S. Wolfman and D. Weld. The LPSAT Engine & its Application to Resource Planning. In *Proc. IJCAI*, 1999.
- [ZM02] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *Proc. CAV'02*, number 2404 in *LNCS*, pages 17–36. Springer, 2002.

Chapter 33

Satisfiability Modulo Theories

Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli

33.1. Introduction

Applications in artificial intelligence and formal methods for hardware and software development have greatly benefited from the recent advances in Boolean satisfiability (SAT). Often, however, applications in these fields require determining the satisfiability of formulas in more expressive logics such as first-order logic. Despite the great progress made in the last twenty years, general-purpose first-order theorem provers (such as provers based on the resolution calculus) are typically not able to solve such formulas directly. The main reason for this is that many applications require not general first-order satisfiability, but rather satisfiability with respect to some *background theory*, which fixes the interpretations of certain predicate and function symbols. For instance, applications using integer arithmetic are not interested in whether there exists a nonstandard interpretation of the symbols $<$, $+$, and 0 that makes the formula

$$x < y \wedge \neg(x < y + 0)$$

satisfiable. Instead, they are interested in whether the formula is satisfiable in an interpretation in which $<$ is the usual ordering over the integers, $+$ is the integer addition function, and 0 is the additive identity. General-purpose reasoning methods can be forced to consider only interpretations consistent with a background theory \mathcal{T} , but only by explicitly incorporating the axioms for \mathcal{T} into their input formulas. Even when this is possible,¹ the performance of such provers is often unacceptable. For some background theories, a more viable alternative is to use reasoning methods tailored to the theory in question. This is particularly the case for *quantifier-free* formulas, first-order formulas with no quantifiers but possibly with variables, such as the formula above.

For many theories, specialized methods actually yield *decision procedures* for the satisfiability of quantifier-free formulas or some subclass thereof. This is the

¹Some background theories such as the theory of real numbers or the theory of finite trees, cannot be captured by a finite set of first-order formulas, or, as in the case of the theory of integer arithmetic (with multiplication), by *any* decidable set of first-order formulas.

case, thanks to classical results in mathematics, for the theory of real numbers and the theory of integer arithmetic (without multiplication). In the last two decades, however, specialized decision procedures have also been discovered for a long and still growing list of other theories with practical applications. These include certain theories of arrays and of strings, several variants of the theory of finite sets or multisets, the theories of several classes of lattices, the theories of finite, regular and infinite trees, of lists, tuples, records, queues, hash tables, and bit-vectors of a fixed or arbitrary finite size.

The research field concerned with the satisfiability of formulas with respect to some background theory is called *Satisfiability Modulo Theories*, or SMT, for short. In analogy with SAT, SMT procedures (whether they are decision procedures or not) are usually referred to as *SMT solvers*. The roots of SMT can be traced back to early work in the late 1970s and early 1980s on using decision procedures in formal methods by such pioneers as Nelson and Oppen [NO80, NO79], Shostak [Sho78, Sho79, Sho84], and Boyer and Moore [BM90, BM88].² Modern SMT research started in the late 1990s with various independent attempts [AG93, GS96, ACG99, PRSS99, BGV99] to build more scalable SMT solvers by exploiting advances in SAT technology. Indeed, the defining characteristic of SMT is that satisfiability of a formula is determined by implicitly or explicitly translating to SAT. For this reason, early SMT solvers were often called *SAT-based decision procedures*. Since 2002, there has been a great deal of interest and research on the foundational and practical aspects of SMT. SMT solvers have been developed in academia and industry with increasing scope and performance. SMT solvers or techniques have been integrated into: interactive theorem provers for high-order logic (such as HOL, Isabelle, and PVS); extended static checkers (such as Boogie and ESC/Java 2); verification systems (such as ACL2, Caduceus, SAL, UCLID, and Why); formal CASE environments (such as KeY); model checkers (such as BLAST, Eureka, MAGIC and SLAM); certifying compilers (such as Touchstone and TVOC); unit test generators (such as DART, EXE, CUTE and PEX).

This chapter provides a brief overview of SMT and its main approaches, together with references to the relevant literature for a deeper study. In particular, it focuses on the two most successful major approaches so far for implementing SMT solvers, usually referred to as the “eager” and the “lazy” approach. We note that in recent years, ideas from both approaches have been combined in successful solvers, but it is still useful from a pedagogical approach to study them separately.

The *eager* approach is based on devising efficient, specialized translations to convert an input formula into an equisatisfiable propositional formula using enough relevant consequences of the theory \mathcal{T} . The approach applies in principle to any theory with a decidable ground satisfiability problem, possibly however at the cost of a significant blow-up in the translation. Its main allure is that the translation imposes upfront all theory-specific constraints on the SAT solver’s search space, potentially solving the input formula quickly; in addition, the translated formula can be given to any off-the-shelf SAT solver. It leverages the ability of modern SAT solvers to quickly process relevant theory-specific information

²Notable early systems building on this work are the Boyer-Moore prover, PVS, Simplify, SteP, and SVC.

encoded into large SAT formulas.

The *lazy* approach consists in building ad-hoc procedures implementing, in essence, an inference system specialized on a background theory \mathcal{T} . The main advantage of theory-specific solvers is that one can use whatever specialized algorithms and data structures are best for the theory in question, which typically leads to better performance. The common practice is to write *theory solvers* just for conjunctions of literals—i.e., atomic formulas and their negations. These pared down solvers are then embedded as separate submodules into an efficient SAT solver, allowing the joint system to accept quantifier-free formulas with an arbitrary Boolean structure.

The rest of this chapter is structured as follows. Section 33.2 provides background information, with formal preliminaries and a brief description of a few theories popular in SMT applications. The next two sections respectively describe the eager and the lazy approach in some detail. Section 33.5 describes some general methods for building theory solvers for the lazy approach. Section 33.6 focuses on techniques for combining solvers for different theories into a solvers for a combination of these theories. Finally, Section 33.7, describes some important extension and enhancements on the methods and techniques described in the previous sections.

33.2. Background

33.2.1. Formal preliminaries

In this chapter we will work in the context of (classical) first-order logic with equality (see, e.g., [EFT94, End00]). To make the chapter more self-contained, however, we introduce here all the relevant basic concepts and notation.

33.2.1.1. Syntax

A *signature* Σ is set of *predicate* and *function* symbols, each with an associated *arity*, a non-negative number. For any signature Σ , we denote by Σ^F and Σ^P respectively the set of function and of predicate symbols in Σ . We call the 0-arity symbols of Σ^F *constant* symbols, and usually denote them by the letters a, b possibly with subscripts. We call the 0-arity symbols of Σ^P *propositional* symbols, and usually denote them by the letters A, B , possibly with subscripts. Also, we use f, g and p, q , possibly with subscripts, to denote respectively the non-constant symbols of Σ^F and the non-propositional symbols of Σ^P .

In this chapter, we are mostly interested in quantifier-free terms and formulas built with the symbols of a given signature Σ . As a technical convenience, we treat the (free) variables of a quantifier-formula as constants in a suitable expansion of Σ . For example, if Σ is the signature of integer arithmetic we consider the formula $x < y + 1$ as a *ground* (i.e., variable-free) formula in which x and y are additional constant symbols. Formally, a *ground* (Σ -)term t and a *ground* (Σ -)formula φ are expressions in the language defined by the abstract grammar in Figure 33.1. As usual, we call *atomic formula* (or *atom*) a formula of the form $A, p(t_1, \dots, t_n)$,

$t ::= c$	where $c \in \Sigma^F$ with arity 0
$ f(t_1, \dots, t_n)$	where $f \in \Sigma^F$ with arity $n > 0$
$ ite(\varphi, t_1, t_2)$	
$\varphi ::= A$	where $A \in \Sigma^P$ with arity 0
$ p(t_1, \dots, t_n)$	where $p \in \Sigma^P$ with arity $n > 0$
$ t_1 = t_2 \mid \perp \mid \top \mid \neg\varphi_1$	
$ \varphi_1 \rightarrow \varphi_2 \mid \varphi_1 \leftrightarrow \varphi_2$	
$ \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2$	

Figure 33.1. Ground terms and formulas

$t_1 = t_2$, \perp , or \top .³ A (Σ) -*literal* is an atomic Σ -formula or the negation of one. We will use the letter l possibly with subscripts, to denote literals. The *complement* of a literal l , written $\neg l$ for simplicity, is $\neg\alpha$ if l is an atomic formula α , and is α if l is $\neg\alpha$. A (Σ) -*clause* is a disjunction $l_1 \vee \dots \vee l_n$ of literals. We will sometimes write a clause in the form of an implication: $\bigwedge_i l_i \rightarrow \bigvee_j l_j$ for $\bigvee_i \neg l_i \vee \bigvee_j l_j$ and $\bigwedge_i l_i \rightarrow \perp$ for $\bigvee_i \neg l_i$ where each l_i and l_j is a positive literal. We denote clauses with the letter c , possibly with subscripts, and identify the empty clause, i.e., the empty disjunction of literals, with the formula \perp . A *unit clause* is a clause consisting of a single literal (not containing \perp or \top). When μ is a finite set of literals l_1, \dots, l_n , we may denote by $\neg\mu$ the clause $\neg l_1 \vee \dots \vee \neg l_n$. Correspondingly, if c is a clause $l_1 \vee \dots \vee l_n$, we denote by $\neg c$ the set $\{\neg l_1, \dots, \neg l_n\}$. A *CNF formula* is a conjunction $c_1 \wedge \dots \wedge c_n$ of zero or more clauses. When it leads to no ambiguities, we will sometimes also write CNF formulas in set notation $\{c_1, \dots, c_n\}$, or simply replace the \wedge connectives by commas.

33.2.1.2. Semantics

Formulas are given a meaning, that is, a truth value from the set $\{\mathbf{true}, \mathbf{false}\}$, by means of (*first-order*) *models*. A model \mathcal{A} for a signature Σ , or Σ -model, is a pair consisting of a non-empty set A , the *universe* of the model, and a mapping $(_)^\mathcal{A}$ assigning to each constant symbol $a \in \Sigma^F$ an element $a^\mathcal{A} \in A$, to each function symbol $f \in \Sigma^F$ of arity $n > 0$ a total function $f^\mathcal{A} : A^n \rightarrow A$, to each propositional symbol $B \in \Sigma^P$ an element $B^\mathcal{A} \in \{\mathbf{true}, \mathbf{false}\}$, and to each $p \in \Sigma^P$ of arity $n > 0$ a total function $p^\mathcal{A} : A^n \rightarrow \{\mathbf{true}, \mathbf{false}\}$. This mapping uniquely determines a homomorphic extension, also denoted as $(_)^\mathcal{A}$, that maps each Σ -term t to an element $t^\mathcal{A} \in A$, and each Σ -formula φ to an element $\varphi^\mathcal{A} \in \{\mathbf{true}, \mathbf{false}\}$. The extension, which we call an *interpretation* of the terms and the formulas, is defined as expected. In particular, for any \mathcal{A} : $f(t_1, \dots, t_n)^\mathcal{A} = f^\mathcal{A}(t_1^\mathcal{A}, \dots, t_n^\mathcal{A})$; $ite(\varphi, t_1, t_2)^\mathcal{A}$ equals $t_1^\mathcal{A}$ if $\varphi^\mathcal{A} = \mathbf{true}$ and $t_2^\mathcal{A}$ otherwise; $p(t_1, \dots, t_n)^\mathcal{A} = p^\mathcal{A}(t_1^\mathcal{A}, \dots, t_n^\mathcal{A})$; $\perp^\mathcal{A} = \mathbf{false}$; $\top^\mathcal{A} = \mathbf{true}$; and $(t_1 = t_2)^\mathcal{A} = \mathbf{true}$ iff $t_1^\mathcal{A} = t_2^\mathcal{A}$.⁴

We say that a Σ -model \mathcal{A} *satisfies* (resp. *falsifies*) a Σ -formula φ iff $\varphi^\mathcal{A}$ is **true**

³Note that, by allowing propositional symbols in signatures this language properly includes the language of propositional logic.

⁴Note that we are using the symbol $=$ both as a symbol of the logic and as the usual meta-symbol for equality. The difference, however, should be always clear from context.

(resp. **false**). In SMT, one is not interested in arbitrary models but in models belonging to a given *theory* \mathcal{T} constraining the interpretation of the symbols of Σ . Following the more recent SMT literature, we define Σ -theories most generally as just sets of one or more (possibly infinitely many) Σ -models. Then, we say that a ground Σ -formula is satisfiable in a Σ -theory \mathcal{T} , or \mathcal{T} -satisfiable, iff there is an element of the set \mathcal{T} that satisfies φ . Similarly, a set Γ of ground Σ -formulas \mathcal{T} -entails a ground formula φ , written $\Gamma \models_{\mathcal{T}} \varphi$, iff every model of \mathcal{T} that satisfies all formulas in Γ satisfies φ as well. We say that Γ is \mathcal{T} -consistent iff $\Gamma \not\models_{\mathcal{T}} \perp$, and that φ is \mathcal{T} -valid iff $\emptyset \models_{\mathcal{T}} \varphi$. We call a clause c a *theory lemma* if c is \mathcal{T} -valid (i.e., $\emptyset \models_{\mathcal{T}} c$). All these notions reduce exactly to the corresponding notions in standard first-order logic by choosing as \mathcal{T} the set of all Σ -models; for that case, we drop \mathcal{T} from the notation (and for instance write just $\Gamma \models \varphi$).

Typically, given a Σ -theory \mathcal{T} , one is actually interested in the \mathcal{T} -satisfiability of ground formulas containing additional, *uninterpreted* symbols, i.e., predicate or function symbols not in Σ . This is particularly the case for uninterpreted constant symbols—which, as we have seen, play the role of free variables—and uninterpreted propositional symbols—which can be used as abstractions of other formulas. Formally, uninterpreted symbols are accommodated in the definitions above by considering instead of \mathcal{T} , the theory \mathcal{T}' defined as follows. Let Σ' be any signature including Σ . An *expansion* \mathcal{A}' to Σ' of a Σ -model \mathcal{A} is a Σ' -model that has the same universe as \mathcal{A} and agrees with \mathcal{A} on the interpretation of the symbols in Σ . The theory \mathcal{T}' is the set of *all possible expansions* of the models of \mathcal{T} to Σ' . To keep the terminology and notation simple, we will still talk about \mathcal{T} -satisfiability, \mathcal{T} -entailment and so on when dealing with formulas containing uninterpreted symbols, but with the understanding that we actually mean the Σ' -theory \mathcal{T}' where Σ' is a suitable expansion of \mathcal{T} 's original signature.

Then, the *ground \mathcal{T} -satisfiability problem* is the problem of determining, given a Σ -theory \mathcal{T} , the \mathcal{T} -satisfiability of ground formulas over an arbitrary expansion of Σ with uninterpreted constant symbols. Since a formula φ is \mathcal{T} -satisfiable iff $\neg\varphi$ is not \mathcal{T} -valid, the ground \mathcal{T} -satisfiability problem has a dual *ground \mathcal{T} -validity problem*. The literature in the field (especially the older literature) sometimes adopts this dual view.⁵ Finally, a theory \mathcal{T} is *convex* if for all sets μ of ground Σ' -literals (where Σ' is an expansion of Σ with uninterpreted constant symbols) and all sets E of equalities between uninterpreted constant symbols in Σ' , $\mu \models_{\mathcal{T}} \bigvee_{e \in E} e$ iff $\mu \models_{\mathcal{T}} e$ for some $e \in E$.

33.2.1.3. Combined Theories

Several applications of SMT deal with formulas involving two or more theories at once. In that case, satisfiability is understood as being modulo some combination of the various theories. If two theories \mathcal{T}_1 and \mathcal{T}_2 are both defined axiomatically, their combination can simply be defined as the theory axiomatized by the union of the axioms of the two theories, \mathcal{T}_1 and \mathcal{T}_2 . This is adequate if the signatures of the two theories are disjoint. If, instead, \mathcal{T}_1 and \mathcal{T}_2 have symbols in common, one has to consider whether a shared function (resp. predicate) symbol is meant

⁵In \mathcal{T} -validity problems, any uninterpreted constant symbols behave like *universally* quantified variables, so it is also common to see ground \mathcal{T} -validity being described in the literature as \mathcal{T} -validity of *universal* Σ -formulas.

to stand for the same function (resp. relation) in each theory or not. In the latter case, a proper signature renaming must be applied to the theories before taking the union of their axioms.

With theories specified as sets of first order models, as done here, a suitable notion of theory combination is defined as follows. Let us say that a Σ -model \mathcal{A} is the Σ -reduct of a Σ' -model \mathcal{B} with $\Sigma' \supseteq \Sigma$ if \mathcal{A} has the same universe as \mathcal{B} and interprets the symbols of Σ exactly as \mathcal{B} does. Then, the *combination* $\mathcal{T}_1 \oplus \mathcal{T}_2$ of \mathcal{T}_1 and \mathcal{T}_2 is the set of all $(\Sigma_1 \cup \Sigma_2)$ -models \mathcal{B} whose Σ_1 -reduct is isomorphic to a model of \mathcal{T}_1 and whose Σ_2 -reduct is isomorphic to a model of \mathcal{T}_2 .⁶ The correspondence with the axiomatic case is given by the following fact (see, e.g., [TR03]): when each \mathcal{T}_i is the set of all Σ_i -models that satisfy some set Γ_i of first-order axioms, $\mathcal{T}_1 \oplus \mathcal{T}_2$ is precisely the set of all $(\Sigma_1 \cup \Sigma_2)$ -models that satisfy $\Gamma_1 \cup \Gamma_2$.

33.2.1.4. Abstraction

For abstraction purposes, we associate with every signature Σ (possibly containing uninterpreted symbols in the sense above) a signature Ω consisting of the propositional symbols of Σ plus a set of new propositional symbols having the same cardinality as the set of ground Σ -atoms. We then fix a bijection $\mathcal{T2B}$, called *propositional abstraction*, between the set of ground Σ -formulas without *ite* expressions and the propositional formulas over Ω . This bijection maps each propositional symbol of Σ to itself and each non-propositional Σ -atom to one of the additional propositional symbols of Ω , and is homomorphic with respect to the logical operators.⁷ The restriction to formulas with no *ite*'s is without loss of generality because *ite* constructs can be eliminated in advance by a satisfiability-preserving transformation that repeatedly applies the following rule to completion: let $ite(\psi, t_1, t_2)$ be a subterm appearing in a formula φ ; we replace this term in φ by some new uninterpreted constant a and return the conjunction of the result with the formula $ite(\psi, a = t_1, a = t_2)$.⁸ We denote by $\mathcal{B2T}$ the inverse of $\mathcal{T2B}$, and call it *refinement*.

To streamline the notation we will often write φ^p to denote $\mathcal{T2B}(\varphi)$. Also, if μ is a set of Σ -formulas, we will write μ^p to denote the set $\{\varphi^p \mid \varphi \in \mu\}$; if μ^p is a set of Boolean literals, then μ will denote $\mathcal{B2T}(\mu^p)$. A Σ -formula φ is *propositionally unsatisfiable* if $\varphi^p \models \perp$. We will often write $\mu \models_p \varphi$ to mean $\mu^p \models \varphi^p$. We point out that for any theory \mathcal{T} , $\mu \models_p \varphi$ implies $\mu \models_{\mathcal{T}} \varphi$, but not vice versa.

⁶We refer the reader to, e.g., [EFT94] for a definition of isomorphic models. Intuitively, two models \mathcal{A} and \mathcal{B} are isomorphic if they are identical with the possible exception that the universe of \mathcal{B} is a renaming of the universe of \mathcal{A} .

⁷That is, $\mathcal{T2B}(\perp) = \perp$, $\mathcal{T2B}(\varphi_1 \wedge \varphi_2) = \mathcal{T2B}(\varphi_1) \wedge \mathcal{T2B}(\varphi_2)$, and so on.

⁸The newly-introduced *ite* can be thought of as syntactic sugar for $\psi \rightarrow a = t_1 \wedge \neg\psi \rightarrow a = t_2$. Alternatively, to avoid potential blowup of the formula, a formula-level if-then-else operator can be introduced into the language syntax.

33.2.2. Some theories of interest

In order to provide some motivation and connection to applications, we here give several examples of theories of interest and how they are used in applications.⁹

33.2.2.1. Equality

As described above, a theory usually imposes some restrictions on how function or predicate symbols may be interpreted. However, the most general case is a theory which imposes no such restrictions, in other words, a theory that includes all possible models for a given signature.

Given any signature, we denote the theory that includes all possible models of that theory as $\mathcal{T}_{\mathcal{E}}$. It is also sometimes called the *empty* theory because its finite axiomatization is just \emptyset (the empty set). Because no constraints are imposed on the way the symbols in the signature may be interpreted, it is also sometimes called the theory of equality with uninterpreted functions (EUF). The satisfiability problem for conjunctions of ground formulas modulo $\mathcal{T}_{\mathcal{E}}$ is decidable in polynomial time using a procedure known as *congruence closure* [BTV03, DST80, NO05b].

Some of the first applications that combined Boolean reasoning with theory reasoning used this simple theory [NO80]. Uninterpreted functions are often used as an abstraction technique to remove unnecessarily complex or irrelevant details of a system being modeled. For example, suppose we want to prove that the following set of literals is unsatisfiable: $\{a * (f(b) + f(c)) = d, b * (f(a) + f(c)) \neq d, a = b\}$. At first, it may appear that this requires reasoning in the theory of arithmetic. However, if we abstract $+$ and $*$ by replacing them with uninterpreted functions g and h respectively, we get a new set of literals: $\{h(a, g(f(b), f(c))) = d, h(b, g(f(a), f(c))) \neq d, a = b\}$. This set of literals can be proved unsatisfiable using only congruence closure.

33.2.2.2. Arithmetic

Let $\Sigma_{\mathcal{Z}}$ be the signature $(0, 1, +, -, \leq)$. Let the theory $\mathcal{T}_{\mathcal{Z}}$ consist of the model that interprets these symbols in the usual way over the integers.¹⁰ This theory is also known as *Presburger arithmetic*.¹¹ We can define the theory $\mathcal{T}_{\mathcal{R}}$ to consist of the model that interprets these same symbols in the usual way over the reals.

Let $\mathcal{T}'_{\mathcal{Z}}$ be the extension of $\mathcal{T}_{\mathcal{Z}}$ with an arbitrary number of uninterpreted constants (and similarly for $\mathcal{T}'_{\mathcal{R}}$). The question of satisfiability for conjunctions of ground formulas in either of these theories is decidable. Ground satisfiability in $\mathcal{T}'_{\mathcal{R}}$ is actually decidable in polynomial time [Kar84], though exponential methods such as those based on simplex often perform best in practice (see, e.g. [DdM06b]). On the other hand, ground $\mathcal{T}'_{\mathcal{Z}}$ -satisfiability is NP-complete [Pap81].

Two important related problems have to do with restricting the syntax of arithmetic formulas in these theories. *Difference logic* formulas require that every

⁹See [MZ03a] for an earlier related survey.

¹⁰Of course, additional symbols can be included for numerals besides 0 and 1 or for $<$, $>$, and \geq , but these add no expressive power to the theory, so we omit them for the sake of simplicity.

¹¹In Presburger arithmetic, the domain is typically taken to be the natural numbers rather than the integers, but it is straightforward to translate a formula with integer variables to one where variables are interpreted over \mathcal{N} and vice-versa by adding (linearly many) additional variables or constraints.

atom be of the form $a - b \bowtie t$ where a and b are uninterpreted constants, \bowtie is either $=$ or \leq , and t is an integer (i.e. either a sum of 1's or the negation of a sum of 1's). Fast algorithms for difference logic formulas have been studied in [NO05c]. A slight variation on difference logic is UTVPI (“unit two variable per inequality”) formulas which in addition to the above pattern also allow $a + b \bowtie t$. Algorithms for UTVPI formulas have been explored in [LM05, SS05].

An obvious extension of the basic arithmetic theories discussed so far is to add multiplication. Unfortunately, this dramatically increases the complexity of the problem, and so is often avoided in practice. In fact, the integer case becomes undecidable even for conjunctions of ground formulas [Mat71]. The real case is decidable but is doubly-exponential [DH88].

There are obviously many practical uses of decision procedures for arithmetic and solvers for these or closely related theories have been around for a long time. In particular, when modeling and reasoning about systems, arithmetic is useful for modeling finite sets, program arithmetic, manipulation of pointers and memory, real-time constraints, physical properties of the environment, etc.

33.2.2.3. Arrays

Let $\Sigma_{\mathcal{A}}$ be the signature $(read, write)$. Let $\Lambda_{\mathcal{A}}$ be the following axioms:

$$\begin{aligned} &\forall a \forall i \forall v (read(write(a, i, v), i) = v) \\ &\forall a \forall i \forall j \forall v (i \neq j \rightarrow read(write(a, i, v), j) = read(a, j)) \end{aligned}$$

Then the theory $\mathcal{T}_{\mathcal{A}}$ of *arrays* is the set of all models of these axioms. It is common also to include the following *axiom of extensionality*:

$$\forall a \forall b ((\forall i (read(a, i) = read(b, i))) \rightarrow a = b)$$

We will denote the resulting theory $\mathcal{T}_{\mathcal{A}ex}$. The satisfiability of ground formulas over $\mathcal{T}'_{\mathcal{A}}$ or $\mathcal{T}'_{\mathcal{A}ex}$ is NP-complete [SDBL01]. Theories of arrays are commonly used to model actual array data structures in programs. They are also often used as an abstraction for memory. The advantage of modeling memory with arrays is that the size of the model depends on the number of accesses to memory rather than the size of the memory being modeled. In many cases, this leads to significant efficiency gains.

33.2.2.4. Fixed-width bit-vectors

A natural theory for high-level reasoning about circuits and programs is a theory of bit-vectors. Various theories of bit-vectors have been proposed and studied [CMR97, Möl97, BDL98, BP98, EKM98, BRTS08]. Typically, constant symbols are used to represent vectors of bits, and each constant symbol has an associated bit-width that is fixed for that symbol. The function and predicate symbols in these theories may include extraction, concatenation, bit-wise Boolean operations, and arithmetic operations. For non-trivial theories of bit-vectors, it is easy to see that the satisfiability problem is NP-complete by a simple reduction to SAT. Bit-vectors provide a more compact representation and often allow

problems to be solved more efficiently than if they were represented at the bit level [BKO⁺07, BCF⁺07, GD07].

33.2.2.5. Inductive data types

An *inductive data type* (IDT) defines one or more *constructors*, and possibly also *selectors* and *testers*. A simple example is the IDT *list*, with constructors *cons* and *null*, selectors *car* and *cdr*, and testers *is_cons* and *is_null*. The *first order signature* of an IDT associates a function symbol with each constructor and selector and a predicate symbol with each tester. The standard model for such a signature is a term model built using only the constructors. For IDTs with a single constructor, a conjunction of literals is decidable in polynomial time using an algorithm by Oppen [Opp80a]. For more general IDTs, the problem is NP complete, but reasonably efficient algorithms exist in practice [BST07]. IDTs are very general and can be used to model a variety of things, e.g., enumerations, records, tuples, program data types, and type systems.

33.3. Eager Encodings to SAT

The *eager approach* to SMT solving involves translating the original formula to an equisatisfiable Boolean formula in a single step. In order to generate a small SAT problem, several optimizations are performed in this translation. As one might expect, some optimizations are computationally expensive and thus there is a trade-off between the degree of optimization performed and the amount of time spent therein. In fact, the translation procedure is much like an optimizing compiler, with the “high-level program” being the original SMT problem and the “low-level object code” being the generated SAT problem. This section describes the main ideas in the eager approach and surveys the state of the art.

33.3.1. Overview

The translations used in the eager approach are, by their very nature, theory-specific. We survey here the transformations for a combination of two theories: $\mathcal{T}_{\mathcal{E}}$ and $\mathcal{T}_{\mathcal{Z}}$. These theories, plus an extension of the basic syntax to include restricted lambda expressions (see below), form the core logic of the original UCLID decision procedure [LS04, Ses05], which is the main tool implementing the eager approach. The UCLID logic has sufficed for a range of applications, from microprocessor design verification [LB03] to analyzing software for security vulnerabilities [Ses05].

As mentioned, the theories of interest for this section are $\mathcal{T}_{\mathcal{E}}$ and $\mathcal{T}_{\mathcal{Z}}$, with signatures as given in §33.2.2. In particular, we assume a signature Σ containing 0, 1, +, −, ≤, and any number of uninterpreted functions and predicates.

Lambda expressions. For additional expressiveness, in this section we consider an extension of the core logical syntax (as given in Figure 33.1) to include lambda expressions. Assuming an infinite set \mathcal{V} of *integer variables*, the extended syntax is given in Figure 33.2.

Notice that the use of lambda expressions is quite restricted. In particular, there is no way in the logic to express any form of iteration or recursion. An

t	$::= c$ $ \textit{int-var}$ $ \textit{function-expr}(t_1, \dots, t_n)$ $ \textit{ite}(\varphi, t_1, t_2)$	where $c \in \Sigma^F$ with arity 0
φ	$::= A$ $ \textit{predicate-expr}(t_1, \dots, t_n)$ $ t_1 = t_2 \mid \perp \mid \top \mid \neg\varphi_1$ $ \varphi_1 \rightarrow \varphi_2 \mid \varphi_1 \leftrightarrow \varphi_2$ $ \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2$	where $A \in \Sigma^P$ with arity 0
$\textit{int-var}$	$::= v$	where $v \in \mathcal{V}$
$\textit{function-expr}$	$::= f$ $ \lambda \textit{int-var}, \dots, \textit{int-var}. t$	where $f \in \Sigma^F$ with arity $n > 0$
$\textit{predicate-expr}$	$::= p$ $ \lambda \textit{int-var}, \dots, \textit{int-var}. \varphi$	where $p \in \Sigma^P$ with arity $n > 0$

Figure 33.2. Extended syntax with restricted lambda expressions.

integer variable x is said to be *bound* in expression E when it occurs inside a lambda expression for which x is one of the argument variables. We say that an expression is *well-formed* when it contains no unbound variables.

Satisfiability and entailment are defined for well-formed formulas just as in §33.2.1 for formulas without lambdas. Well-formed formulas containing lambdas are considered to be semantically equivalent to their beta-reduced forms (see §33.3.1.2, below), which do not contain lambdas.

Lambda notation allows us to model the effect of a sequence of *read* and *write* operations on a memory (the *select* and *update* operations on an array) without the added complexity of the theory of arrays (\mathcal{T}_A). At any point of system operation, a memory can be represented by a function expression M denoting a mapping from addresses to values (for an array, the mapping is from indices to values). The initial state of the memory is given by an uninterpreted function symbol m_0 indicating an arbitrary memory state. The effect of a write operation with terms A and D denoting the address and data values yields a function expression M' :

$$M' = \lambda \textit{addr}. \textit{ITE}(\textit{addr} = A, D, M(\textit{addr}))$$

Reading from array M at address A simply yields the function application $M(A)$.

Multi-dimensional memories or arrays are easily expressed in the same way. Moreover, lambda expressions can express *parallel-update* operations, which update multiple memory locations in a single step. The details are outside the scope of this chapter, and can be found elsewhere [Ses05].

33.3.1.1. Operation

Suppose that we are given a formula F_{orig} in the syntax of Figure 33.2. We decide the satisfiability of F_{orig} by performing a three-stage satisfiability-preserving translation to a Boolean formula F_{bool} , and then invoking a SAT solver on F_{bool} .

The three stages of translation are as follows:

1. All lambda expressions are eliminated, resulting in a formula F_{norm} . This stage is described in §33.3.1.2.
2. Function and predicate applications of non-zero arity are eliminated to get a formula F_{arith} . This stage is described in §33.3.1.3.
3. Formula F_{arith} is a quantifier-free linear integer arithmetic (Σ_Z -) formula. There is more than one way to translate F_{arith} to an equisatisfiable Boolean formula F_{bool} . We describe these techniques in §33.3.2–§33.3.4.

In addition to preserving satisfiability, the mapping between the eliminated symbols of a theory and the new symbols that replace them is maintained. For satisfiable problems, this facilitates model generation from a satisfying assignment generated by the SAT solver. For unsatisfiable problems, it permits the generation of higher level proof information from a Boolean proof of unsatisfiability.

33.3.1.2. Eliminating Lambdas

Recall that the syntax of lambda expressions does not permit recursion or iteration. Therefore, each lambda application in F_{orig} can be expanded by *beta-substitution*, i.e., by replacing each argument variable with the corresponding argument term. Denote the resulting formula by F_{norm} .

This step can result in an exponential blow-up in formula size. Suppose that all expressions in our logic are represented as directed acyclic graphs (DAGs) so as to share common sub-expressions. Then, the following example shows how we can get an exponential-sized DAG representation of F_{norm} starting from a linear-sized DAG representation of F_{orig} .

Example 33.3.1. Let F_{orig} be defined recursively by the following set of expressions:

$$\begin{aligned}
 F_{orig} &\doteq P(L_1(b)) \\
 L_1 &\doteq \lambda x . f_1(L_2(x), L_2(g_1(x))) \\
 L_2 &\doteq \lambda x . f_2(L_3(x), L_3(g_2(x))) \\
 &\vdots \\
 L_{n-1} &\doteq \lambda x . f_{n-1}(L_n(x), L_n(g_{n-1}(x))) \\
 L_n &\doteq g_n
 \end{aligned}$$

Notice that the representation of F_{orig} is linear in n . Suppose we perform beta-substitution on L_1 . As a result, the sub-expression $L_1(b)$ gets transformed to $f_1(L_2(b), L_2(g_1(b)))$. Next, if we expand L_2 , we get four applications of L_3 , viz., $L_3(b)$, $L_3(g_1(b))$, $L_3(g_2(b))$, and $L_3(g_2(g_1(b)))$. Notice that there were originally only two applications of L_3 .

Continuing the elimination process, after $k - 1$ elimination steps, we will get 2^{k-1} distinct applications of L_k . This can be formalized by observing that after $k - 1$ steps each argument to L_k is comprised of applications of functions from a distinct subset of $\mathcal{P}(\{g_1, g_2, \dots, g_{k-1}\})$. Thus, after all lambda elimination steps, F_{norm} will contain 2^{n-1} distinct applications of g_n , and hence is exponential in the size of F_{orig} . \square

In practice, however, this exponential blow-up is rarely encountered. This is because the recursive structure in most lambda expressions, including those for memory (array) operations, tends to be linear. For example, here is the lambda expression corresponding to the result of the memory write (store) operation:

$$\lambda addr . ITE(addr = A, D, M(addr))$$

Notice that the “recursive” use of M occurs only in one branch of the ITE expression.

33.3.1.3. Eliminating Function Applications

The second step in the transformation to a Boolean formula is to eliminate applications of function and predicate symbols of non-zero arity. These applications are replaced by symbolic constants, but only after encoding enough information to maintain functional consistency (the congruence property).

There are two different techniques of eliminating function (and predicate) applications. The first is a classic method due to Ackermann [Ack54] that involves creating sufficient instances of the congruence axiom to preserve satisfiability. The second is a technique introduced by Bryant et al. [BGV01] that exploits the polarity of equations and is based on the use of ITE expressions. We briefly review each of these methods.

Ackermann’s method. We illustrate Ackermann’s method using an example. Suppose that function symbol f has three occurrences: $f(a_1)$, $f(a_2)$, and $f(a_3)$. First, we generate three fresh symbolic constants xf_1 , xf_2 , and xf_3 to replace all instances of these applications in F_{norm} .

Then, the following set of functional consistency constraints for f is generated:

$$\left\{ a_1 = a_2 \implies xf_1 = xf_2, \quad a_1 = a_3 \implies xf_1 = xf_3, \quad a_2 = a_3 \implies xf_2 = xf_3 \right\}$$

In a similar fashion, functional consistency constraints are generated for each function and predicate symbol in F_{norm} . Denote the conjunction of all these constraints by F_{cong} . Then, F_{arith} is the formula $F_{cong} \wedge F_{norm}$.

The Bryant-German-Velev method. The function elimination method proposed by Bryant, German, and Velev exploits a property of function applications called *positive equality* [BGV01]. (This discussion assumes that we are working with the concept of validity rather than satisfiability, but the ideas remain unchanged except for a flipping of polarities.) The general idea is to determine the polarity of each equation in the formula, i.e., whether it appears under an even (positive) or odd (negative) number of negations. Applications of uninterpreted functions can then be classified as either p-function applications, i.e., used only under positive equalities, or g-function applications, i.e., general function applications that appear under other equalities or under inequalities. The p-function applications can be encoded in propositional logic with fewer Boolean variables than the g-function applications, thus greatly simplifying the resulting SAT problem. We omit the details.

In order to exploit positive equality, Bryant et al. eliminate function applications using a nested series of ITE expressions. As an example, if function symbol

f has three occurrences: $f(a_1)$, $f(a_2)$, and $f(a_3)$, then we would generate three new symbolic constants xf_1 , xf_2 , and xf_3 . We would then replace all instances of $f(a_1)$ by xf_1 , all instances of $f(a_2)$ by $ITE(a_2 = a_1, xf_1, xf_2)$, and all instances of $f(a_3)$ by $ITE(a_3 = a_1, xf_1, ITE(a_3 = a_2, xf_2, xf_3))$. It is easy to see that this preserves functional consistency.

Predicate applications can be removed by a similar process. In eliminating applications of some predicate p , we introduce symbolic Boolean constants xp_1, xp_2, \dots . Function and predicate applications in the resulting formula F_{arith} are all of zero arity.

Lahiri et al. [LBGT04] have generalized the notion of positive equality to apply to both polarities and demonstrate that further optimization is possible, albeit at the cost of incurring a time overhead.

33.3.1.4. Summary

We conclude this section with observations on the worst-case blow-up in formula size in going from the starting formula F_{orig} to the quantifier-free arithmetic formula F_{arith} . The lambda elimination step can result in a worst-case exponential blow-up, but is typically only linear. In going from the lambda-free formula F_{norm} to F_{arith} , the worst-case blow-up is only quadratic. Thus, if the result of lambda expansion is linear in the size of F_{orig} , F_{arith} is at most quadratic in the size of F_{orig} .

We next consider the two main classes of methods for transforming F_{arith} to an equisatisfiable Boolean formula F_{bool} : the *small-domain encoding* method and the *direct encoding* method.

33.3.2. Small-domain encodings

The formula F_{arith} is a quantifier-free $\Sigma_{\mathcal{Z}}$ -formula, and so, we are concerned with $\mathcal{T}_{\mathcal{Z}}$ -satisfiability of quantifier-free $\Sigma_{\mathcal{Z}}$ -formulas. Recall that this problem is NP-complete (as discussed in 33.2.2). In the remainder of this section, we assume satisfiability is with respect to $\mathcal{T}_{\mathcal{Z}}$ and that all formulas are quantifier-free $\Sigma_{\mathcal{Z}}$ -formulas.

A formula is constructed by combining linear constraints with Boolean operators (such as \wedge , \vee , \neg). Formally, the i^{th} constraint is of the form

$$\sum_{j=1}^n a_{i,j} x_j \geq b_i$$

where the coefficients and the constant terms are integer constants and the variables¹² x_1, x_2, \dots, x_n are integer-valued.

If there is a satisfying solution to a formula, there is one whose size, measured in bits, is polynomially bounded in the problem size [BT76, vzGS78, KM78, Pap81]. Problem size is traditionally measured in terms of the parameters m , n , $\log a_{\max}$, and $\log b_{\max}$, where m is the total number of constraints in the formula, n is the number of variables (integer-valued symbolic constants), and

¹²The word “variable” is used in this section instead of “symbolic constant” as it is the more common term used in literature on solving integer linear arithmetic constraints.

$a_{\max} = \max_{(i,j)} |a_{i,j}|$ and $b_{\max} = \max_i |b_i|$ are the maximums of the absolute values of coefficients and constant terms respectively.

The above result implies that we can use an enumerative approach to deciding the satisfiability of a formula, where we restrict our search for satisfying solutions to within the bound on the problem size mentioned above. This approach is referred to as the *small-domain encoding* (SD) method.

In this method, given a formula $F_{\mathcal{Z}}$, we first compute the polynomial bound S on solution size, and then search for a satisfying solution to $F_{\mathcal{Z}}$ in the bounded space $\{0, 1, \dots, 2^S - 1\}^n$. However, a naïve implementation of a SD-based decision procedure fails for formulas encountered in practice. The problem is that the bound on solution size, S , is $\mathcal{O}(\log m + \log b_{\max} + m[\log m + \log a_{\max}])$. In particular, the presence of the $m \log m$ term means that, for problems involving thousands of constraints and variables, as often arises in practice, the Boolean formulas generated are beyond the capacity of even the best current SAT solvers.

In this section, we describe how the small-domain encoding method can be made practical by considering special cases of formulas. Since we consider arbitrary Boolean combinations of these constraints, the decision problems are NP-complete no matter how simple the form of the linear constraint.

33.3.2.1. Equalities

When all linear constraints are equalities (or disequalities) over integer variables, the fragment of $\mathcal{T}_{\mathcal{Z}}$ is called *equality logic*. For this fragment, we have the following “folk theorem” that is easily obtained:

Theorem 33.3.2. For an equality logic formula with n variables, $S = \log n$.

The key proof argument is that any satisfying assignment can be translated to the range $\{0, 1, 2, \dots, n - 1\}$, since we can only tell *whether* variable values differ, not by how much.

This bound yields a search space of size $\mathcal{O}(n^n)$, which can be far too large in many cases. Several optimizations are possible. The most obvious is to divide the set of variables into equivalence classes so that two variables that appear in the same equality fall into the same class. Then a separate bound on solution size can be derived for each equivalence class. This optimization is clearly applicable for arbitrary linear constraints, not just equalities.

The *range allocation* method [PRSS02] is another effective technique for reducing the size of the search space. This method operates by first building a constraint graph representing equalities and disequalities between variables in the formula, with separate types of edges representing equalities and disequalities in the negation normal form. Connected components of this graph correspond to equivalence classes mentioned above. Furthermore, the only major restriction on satisfying solutions comes from cycles that contain exactly one disequality edge. Pnueli et al. [PRSS02] give a graph-based algorithm that assigns, to each variable that participates in both equality and disequality constraints, a set of values large enough to consider all possible legal truth assignments to the constraints containing it. The resulting set of values assigned to each variable can be of cardinality smaller than n , thus often resulting in a more compact search space than n^n .

However, the worst case solution size is still n , and the search space can still be $\Theta(n^n)$.

33.3.2.2. Difference Logic

Recall that *difference logic* requires every atom to be of the form $x_i - x_j \bowtie b_t$ where x_i and x_j are variables, \bowtie is either $=$ or \leq , and b_t is an integer. Note that a constraint of the form $x_i \bowtie b_t$ can be written as $x_i - x_0 \bowtie b_t$ where x_0 is a special “variable” denoting zero. Also note that an equality can be written as a conjunction of two inequalities, while a strict inequality $<$ can be rewritten as a *le* inequality by rounding the constant term down.

We will refer to such atoms as *difference constraints*. They are also referred to in the literature as *difference-bound constraints* or *separation predicates*, and difference logic has also been termed as *separation logic*. We will use DL as an acronym for difference logic.

A fundamental construct used in the SAT-encoding of difference logic is the *constraint graph*. This graph is a weighted, directed multigraph built from the set of m difference constraints involving n variables as follows:

1. A vertex v_i is introduced for each variable x_i , including for x_0 .
2. For each difference constraint of the form $x_i - x_j \geq b_t$, we add a directed edge from v_i to v_j of weight b_t .

The resulting structure has m edges and $n + 1$ vertices. It is, in general, a multigraph since there can be multiple constant (right-hand side) terms for a given left-hand side expression $x_i - x_j$.

The following theorem states the bound on solution size for difference logic.

Theorem 33.3.3. Let F_{diff} be a DL formula with n variables, excluding x_0 . Let b_{max} be the maximum over the absolute values of all difference constraints in F_{diff} . Then, F_{diff} is satisfiable if and only if it has a solution in $\{0, 1, 2, \dots, d\}^n$ where $d = n \cdot (b_{max} + 1)$.

The proof can be obtained by an analysis of the constraint graph \mathcal{G} . The main insight is that any satisfying assignment for a formula with constraints represented by \mathcal{G} can have a spread in values that is at most the weight of the longest path in \mathcal{G} . This path weight is at most $n \cdot (b_{max} + 1)$. The bound is tight, the “+1” in the second term arising from a “rounding” of inequalities from strict to non-strict.

The above bound can also be further optimized. Equivalence classes can be computed just as before. The range allocation approach has also been extended to apply to difference logic, although the analysis involved in computing ranges can take worst-case exponential time [TSSP04]. On the plus side, the range allocation approach can, in some cases, exponentially reduce the size of the search space.

33.3.2.3. UTVPI Constraints

UTVPI constraints include difference constraints as well as *sum constraints* of the form $x_i + x_j \bowtie b_t$. UTVPI constraints over integer variables are also called *generalized 2SAT constraints*. Useful optimization problems, such as the minimum vertex cover and the maximum independent set problems, can be modeled

using UTVPI constraints [HMNT93], and some applications of constraint logic programming and automated theorem proving also generate UTVPI constraints (e.g., see [JMSY94, BCLZ04]).

The bound on solution size for UTVPI constraints is only a slight increase over that for difference logic.

Theorem 33.3.4. Let F_{utvpi} be a UTVPI formula with n variables. Let b_{\max} be the maximum over the absolute values of all constraints in F_{utvpi} . Then, F_{utvpi} is satisfiable if and only if it has a solution in $\{0, 1, 2, \dots, d\}^n$ where $d = 2 \cdot n \cdot (b_{\max} + 1)$.

The theorem can be proved using results from polyhedral theory [Ses05, SSB07]. A UTVPI formula can be viewed as a union of polyhedra defined by UTVPI hyperplanes. The vertices of these polyhedra are half-integral. The main step in the proof is to show that if a satisfying solution exists (i.e., there is an integer point inside some polyhedron in the union), then there is a solution that can be obtained by rounding some vertex of a polyhedron in the union. Since the above bound works for the vertices of the UTVPI polyhedra, it suffices for searching for integer solutions also.

33.3.2.4. Sparse, Mostly-Difference Constraints

The most general case is when no restricting assumptions can be made on the structure of linear constraints. In this case, we can still improve the “typical-case” complexity of SAT-encoding by exploiting the structure of constraints that appear in practical problems of interest.

It has been observed [Ses05, Pra77, DNS03] that formulas arising in software verification have:

1. *Mainly Difference Constraints:* Of the m constraints, $m - k$ are difference constraints, where $k \ll m$.
2. *Sparse Structure:* The k non-difference constraints are sparse, with at most w variables per constraint, where w is “small”. The parameter w is termed the *width* of the constraint.

Seshia and Bryant [SB05] exploited above special structure to obtain a bound on solution size that is parameterized in terms of k and w in addition to m , n , a_{\max} and b_{\max} . Their main result is stated in the following theorem.

Theorem 33.3.5. Let $F_{\mathcal{Z}}$ be a quantifier-free $\Sigma_{\mathcal{Z}}$ -formula. If $F_{\mathcal{Z}}$ is satisfiable, there is a solution to $F_{\mathcal{Z}}$ whose l_{∞} norm is bounded by $d = (n + 2)\Delta$, where $\Delta = s(b_{\max} + 1)(a_{\max} w)^k$ and $s = \min(n + 1, m)$.

The proof of the above theorem is based on a theorem given by Borosh, Treybig, and Flahive [BFT86] bounding integer solutions of linear systems of the form $A\mathbf{x} \geq \mathbf{b}$, where \mathbf{x} is of length n . Their result is as follows. Consider the augmented matrix $[A|\mathbf{b}]$. Let Δ be the maximum of the absolute values of all minors of this augmented matrix. Then, the linear system has a satisfying solution if and only if it has one with all entries bounded by $(n + 2)\Delta$. Seshia and Bryant used the special structure of sparse, mostly-difference constraints to obtain a bound on the value of Δ .

Several optimizations are possible to reduce the size of the bound given in Theorem 33.3.5, including computing equivalence classes of variables, rewriting constraints to reduce the size and number of non-zero coefficients, a “shift-of-origin” transformation to deal with large constant terms, etc. For brevity, these are omitted here and can be found elsewhere [Ses05].

33.3.2.5. Summary

Table 33.1 summarizes the value of d for all the classes of linear constraints explored in this section. We can clearly see that the solution bound for arbitrary

Table 33.1. Solution bounds for classes of linear constraints. The classes are listed top to bottom in increasing order of expressiveness.

Class of Linear Constraints	Solution Bound d
Equality constraints	n
Difference constraints	$n \cdot (b_{\max} + 1)$
UTVPI constraints	$2 \cdot n \cdot (b_{\max} + 1)$
Arbitrary linear constraints	$(n + 2) \cdot \min(n + 1, m) \cdot (b_{\max} + 1) \cdot (w \cdot a_{\max})^k$

formulas is conservative. For example, if all constraints are difference constraints, the expression for d simplifies to $(n + 2) \cdot \min(n + 1, m) \cdot (b_{\max} + 1)$. This is $n + 2$ times as big as the bound obtainable for difference logic in isolation; however, the slack in the bound is a carry-over from the result of Borosh, Treybig, and Flahive [BFT86]. For UTVPI constraints too, the bound derived for arbitrary formulas is much looser. In the worst case, it is looser by an exponential factor: if k is $\mathcal{O}(m)$, a_{\max} is 1, and w is 2, then the bound is $\mathcal{O}((n + 2) \cdot \min(n + 1, m) \cdot (b_{\max} + 1) \cdot 2^m)$, whereas the results of §33.3.2.3 tell us that the solution bound $d = 2 \cdot n \cdot (b_{\max} + 1)$ suffices.

33.3.3. Direct encoding of theory axioms

A decision procedure based on the direct encoding method operates in three steps:

1. Replace each unique constraint in the linear arithmetic formula F_{arith} with a fresh Boolean variable to get a Boolean formula F_{bvar} .
2. Generate a Boolean formula F_{cons} that constrains values of the introduced Boolean variables so as to preserve the arithmetic information in F_{arith} .
3. Invoke a SAT solver on the Boolean formula $F_{bvar} \wedge F_{cons}$.

The direct encoding approach has also been termed as *per-constraint encoding*.

For integer linear arithmetic, the formula F_{cons} expresses so-called *transitivity constraints*. For equality logic, Bryant and Velev [BV00] showed how to generate transitivity constraints efficiently so that the size of F_{cons} is in the worst-case only cubic in the size of the number of equalities in F_{arith} . Their transitivity constraint generation algorithm operates on the constraint graph (as introduced in §33.3.2.1). It avoids enumeration of cycles by introducing *chordal edges*, thereby avoiding an exponential blow-up in the number of transitivity constraints.

Strichman et al. [SSB02] generalized the above graph-based approach to difference logic. In this case, the constraint graph is just as in §33.3.2.2. Again, cycle enumeration is avoided by the introduction of new edges. However, in the case of difference logic, the number of added edges can be exponential in the original constraint graph, in the worst case [Ses05]. Even so, in many practical instances, the number of transitivity constraints is small and the resulting SAT problem is easily solved. Various heuristic optimizations are possible based on the Boolean structure of the formula [Str02b].

Strichman [Str02a] also extended the above scheme to operate on an arbitrary linear arithmetic formula (over the integers or the rationals). The “transitivity constraints” are generated using the Fourier-Motzkin variable elimination procedure (the Omega test variant [Pug91], in the case of integers). It is well-known that Fourier-Motzkin can generate doubly-exponentially many new constraints in the worst case [Cha93]. Thus, the worst-case size of F_{cons} is doubly-exponential in the size of F_{arith} . This worst-case behavior does seem to occur in practice, as evidenced by the results in Strichman’s paper [Str02a] and subsequent work.

We summarize the complexity of the direct encoding method for the three classes of linear arithmetic formulas in Table 33.2.

Table 33.2. Worst-case size of direct encoding. The classes are listed top to bottom in increasing order of expressiveness.

Class of Linear Constraints	Worst-case size of F_{cons}
Equality constraints	Cubic
Difference constraints	Exponential
Arbitrary linear constraints	Doubly exponential

33.3.4. Hybrid eager approaches

In §33.3.2 and §33.3.3, we have introduced two very distinct methods of deciding a linear arithmetic formula via translation to SAT. This naturally gives rise to the following question: Given a formula, which encoding technique should one use to decide *that* formula the fastest? This question is an instance of the automated algorithm selection problem.

On first glance, it might seem that the small-domain encoding would be best, since it avoids the potential exponential or doubly-exponential blowup in SAT problem size that the direct encoding can suffer in the worst case. However, this blowup is not always a problem because of the special structure of the generated SAT instance. The form of a transitivity constraint is $b_1 \wedge b_2 \implies b_3$ where b_1, b_2, b_3 are Boolean variables encoding linear constraints. If the polarities of these variables are chosen appropriately, the resulting constraint is either a Horn constraint or can be transformed into one by variable renaming. Thus, the overall SAT encoding is a “mostly-HornSAT” problem: i.e., the vast majority of clauses are Horn clauses. It has been observed for difference logic that the generated SAT problems are solved quickly in practice in spite of their large size [Ses05].

The question on the choice of encoding has been studied in the context of difference logic [Ses05]. It has been found that this question cannot be resolved entirely in favor of either method. One can select an encoding method based on formula characteristics using a rule generated by machine learning from past examples (formulas) [SLB03, Ses05]. Moreover, parts of a single formula corresponding to different variable classes can be encoded using different encoding methods. The resulting hybrid encoding algorithm has been empirically shown to be more robust to variation in formula characteristics than either of the two techniques in isolation [Ses05]. This hybrid algorithm is the first successful use of automated algorithm selection based on machine learning in SMT solvers.

33.3.5. Eager encoding for bit-vector arithmetic

The theory of fixed-width (finite-precision) bit-vector arithmetic naturally lends itself to the eager encoding approach. The obvious encoding to SAT involves replacing each bit-vector variable in the SMT formula by a vector of Boolean variables, and each bit-vector operator by a corresponding bit-level circuit implementation, thus reducing the original SMT formula to an equi-satisfiable Boolean formula. This approach is commonly termed *bit-blasting*. Bit-blasting remains a workhorse of SMT solving for bit-vector arithmetic, but it does not suffice to solve the more challenging bit-vector formulas. Therefore, researchers have devised several strategies for solving bit-vector formulas before resorting to bit-blasting, or to enhance the effectiveness of bit-blasting. The UCLID solver demonstrated the use of under-approximations of the domains of bit-vector variables (similar to small-domain encoding) and over-approximations of the bit-vector constraints (similar to a version of the direct encoding) to effectively solve bit-vector formulas [BKO⁺07]. The Beaver SMT solver [JLS09], developed in the UCLID project, uses a combination of event-driven bit-vector constraint propagation, specialized bit-vector rewrite rules, optimized encodings of non-linear bit-vector operations, and the use of logic optimization methods on the bit-blasted translations. Boolector [BB09] solves a combination of bit-vectors and arrays by combining rewrite rules, under-approximations, bit-blasting and on-demand checking of consistency with the theory of arrays. Boolector can also reason about combinations of bit-vectors, arrays, and lambda expressions using similar techniques [PNB15]. It has also been found that combining eager and lazy encoding approaches can be effective in the solving of bit-vector arithmetic [BKO⁺07, HBJ⁺14].

33.4. Integrating Theory Solvers into SAT Engines

The alternative to the eager approach, as described above, is the *lazy* approach in which efficient SAT solvers are integrated with decision procedures for first-order theories (or *Theory Solvers* or *T-solvers*) [AG93, WW99, ACG99, dMRS02, ABC⁺02, BDS02a, GHN⁺04, NO05c, YM06]). Systems based on this approach are called *lazy SMT solvers*, and include ArgoLib [MJ04], Ario [SS05], Barcelogic [NO05a], CVC3 [BT07], Fx7 [ML06], ICS [FORS01], MATHSAT [BBC⁺05a], Simplify [DNS03], TSAT++ [ACGM04], Verifun [FJOS03], YICES [DdM06a], and Z3 [dMB08].

33.4.1. Theory Solvers and their desirable features

In its simplest form, a theory solver for a theory \mathcal{T} (\mathcal{T} -solver) is a procedure which takes as input a collection of \mathcal{T} -literals μ and decides whether μ is \mathcal{T} -satisfiable. In order for a \mathcal{T} -solver to be effectively used within a lazy SMT solver, the following features are often important or even essential. In the following, we assume an SMT solver has been called on a \mathcal{T} -formula φ .

Model generation: when the \mathcal{T} -solver is invoked on a \mathcal{T} -consistent set μ , it is able to produce a \mathcal{T} -model \mathcal{I} witnessing the consistency of μ , i.e., $\mathcal{I} \models_{\mathcal{T}} \mu$.

Conflict set generation: when the \mathcal{T} -solver is invoked on a \mathcal{T} -inconsistent set μ , it is able to produce the (possibly minimal) subset η of μ which has caused its inconsistency. η is called a *theory conflict set* of μ .

Incrementality: the \mathcal{T} -solver “remembers” its computation status from one call to the next, so that, whenever it is given as input a set $\mu_1 \cup \mu_2$ such that μ_1 has just been proved \mathcal{T} -satisfiable, it avoids restarting the computation from scratch.

Backtrackability: it is possible for the \mathcal{T} -solver to undo steps and return to a previous state in an efficient manner.

Deduction of unassigned literals: when the \mathcal{T} -solver is invoked on a \mathcal{T} -consistent set μ , it can also perform deductions of the form $\eta \models_{\mathcal{T}} l$, where $\eta \subseteq \mu$ and l is a literal on a not-yet-assigned atom in φ .¹³

Deduction of interface equalities: when returning **Sat**, the \mathcal{T} -solver can also perform deductions of the form $\mu \models_{\mathcal{T}} e$ (if \mathcal{T} is convex) or $\mu \models_{\mathcal{T}} \bigvee_j e_j$ (if \mathcal{T} is not convex) where e, e_1, \dots, e_n are equalities between variables or terms occurring in atoms in μ . We call such equalities *interface equalities* and denote the interface equality ($v_i = v_j$) by e_{ij} . Deductions of interface equalities are also called *e_{ij} -deductions*. Notice that here the deduced equalities need not occur in the input formula φ .

\mathcal{T} -solvers will be discussed in more detail in §33.5.

33.4.2. A generalized DPLL schema

Different variants of lazy SMT procedures have been presented. Here we consider variants in which the Boolean abstraction φ^p of the input formula φ is fed into a DPLL-based SAT solver (henceforth referred to as a DPLL solver). Note that since most DPLL solvers accept only CNF, if φ^p has arbitrary Boolean structure, it is first converted into an equisatisfiable CNF formula using standard techniques (see Chapter 2).

In its simplest integration schema [BDS02a, dMRS02], called “*offline*” schema [FJOS03],¹⁴ the Boolean abstraction φ^p of the input formula is fed to a DPLL solver, which either decides that φ^p is unsatisfiable, and hence φ is \mathcal{T} -unsatisfiable, or it returns a satisfying assignment μ^p ; in the latter case, the set of literals μ

¹³Notice that, in principle, every \mathcal{T} -solver has deduction capabilities, as it is always possible to call \mathcal{T} -solver($\mu \cup \{-l\}$) for every unassigned literal l [ACG99]. We call this technique *plunging* [DNS03]. In practice, plunging is very inefficient.

¹⁴The offline approach is also called the “lemmas on demand” approach in [dMRS02].

```

1.  SatValue  $\mathcal{T}$ -DPLL ( $\mathcal{T}$ -formula  $\varphi$ ,  $\mathcal{T}$ -assignment &  $\mu$ ) {
2.      if ( $\mathcal{T}$ -preprocess( $\varphi, \mu$ ) == Conflict);
3.      return Unsat;
4.       $\varphi^p = \mathcal{T}2\mathcal{B}(\varphi)$ ;  $\mu^p = \mathcal{T}2\mathcal{B}(\mu)$ ;
5.      while (1) {
6.           $\mathcal{T}$ -decide_next_branch( $\varphi^p, \mu^p$ );
7.          while (1) {
8.              status =  $\mathcal{T}$ -deduce( $\varphi^p, \mu^p$ );
9.              if (status == Sat) {
10.                  $\mu = \mathcal{B}2\mathcal{T}(\mu^p)$ ;
11.                 return Sat; }
12.             else if (status == Conflict) {
13.                 blevel =  $\mathcal{T}$ -analyze_conflict( $\varphi^p, \mu^p$ );
14.                 if (blevel == 0)
15.                     return Unsat;
16.                 else  $\mathcal{T}$ -backtrack(blevel,  $\varphi^p, \mu^p$ );
17.             }
18.             else break;
19.         } } }

```

Figure 33.3. An online schema for \mathcal{T} -DPLL based on modern DPLL.

corresponding to μ^p is given as input to the \mathcal{T} -solver. If μ is found to be \mathcal{T} -consistent, then φ is \mathcal{T} -consistent. If not, $\neg\mu^p$ is added as a clause to φ^p , and the SAT solver is restarted from scratch on the resulting formula. Notice that here DPLL is used as a black-box.

In a more sophisticated schema [AG93, GS96, ACG99, WW99, ABC⁺02, FJOS03, GHN⁺04, BBC⁺06], called “*online*” [FJOS03], DPLL is modified to work directly as an enumerator of truth assignments, whose \mathcal{T} -satisfiability is checked by a \mathcal{T} -solver. This schema evolved from that of the DPLL-based procedures for modal logics (see Sections 32.3 and 32.4). Figure 33.3 shows an online \mathcal{T} -DPLL procedure based on a modern DPLL engine [ZMMM01, ZM02]. The inputs φ and μ are a \mathcal{T} -formula and a reference to an (initially empty) set of \mathcal{T} -literals respectively. The DPLL solver embedded in \mathcal{T} -DPLL reasons on and updates φ^p and μ^p , and \mathcal{T} -DPLL maintains some data structure encoding the set $Lits(\varphi)$ and the bijective mapping $\mathcal{T}2\mathcal{B}/\mathcal{B}2\mathcal{T}$ on literals.¹⁵

\mathcal{T} -preprocess simplifies φ into a simpler formula, and updates μ if necessary, so as to preserve the \mathcal{T} -satisfiability of $\varphi \wedge \mu$. If this process produces some conflict, then \mathcal{T} -DPLL returns Unsat. \mathcal{T} -preprocess combines most or all of the Boolean preprocessing steps of DPLL with some theory-dependent rewriting steps on the \mathcal{T} -literals of φ . (The latter are described in §33.4.3.1. and §33.4.3.2.)

\mathcal{T} -decide_next_branch selects the next literal to split on as in standard DPLL (but it may also take into consideration the semantics in \mathcal{T} of the literals being selected.)

\mathcal{T} -deduce, in its simplest version, behaves similarly to standard BCP in DPLL: it iteratively deduces Boolean literals l^p implied by the current assignment

¹⁵We implicitly assume that all functions called in \mathcal{T} -DPLL have direct access to $Lits(\varphi)$ and to $\mathcal{T}2\mathcal{B}/\mathcal{B}2\mathcal{T}$, and that both $\mathcal{T}2\mathcal{B}$ and $\mathcal{B}2\mathcal{T}$ require constant time for mapping each literal.

(i.e., s.t. $\varphi^p \wedge \mu^p \models_p l^p$, “ \models_p ” being propositional entailment) and updates φ^p and μ^p accordingly, until one of the following conditions occur:

- (i) μ^p propositionally violates φ^p ($\mu^p \wedge \varphi^p \models_p \perp$). If so, \mathcal{T} -deduce behaves like deduce in DPLL, returning Conflict.
- (ii) μ^p propositionally satisfies φ^p ($\mu^p \models_p \varphi^p$). If so, \mathcal{T} -deduce invokes the \mathcal{T} -solver on μ : if the latter returns Sat, then \mathcal{T} -deduce returns Sat; otherwise, \mathcal{T} -deduce returns Conflict.
- (iii) no more literals can be deduced. If so, \mathcal{T} -deduce returns Unknown. A slightly more elaborate version of \mathcal{T} -deduce can invoke the \mathcal{T} -solver on μ also at this intermediate stage: if the \mathcal{T} -solver returns Unsat, then \mathcal{T} -deduce returns Conflict. (This enhancement, called *early pruning*, is discussed in §33.4.3.3.)

A much more elaborate version of \mathcal{T} -deduce can be implemented if the \mathcal{T} -solver is able to perform deductions of unassigned literals. (This enhancement, called \mathcal{T} -propagation, is discussed in §33.4.3.4.)

\mathcal{T} -analyze_conflict is an extension of analyze_conflict of DPLL [ZM02, ZMMM01]: if the conflict produced by \mathcal{T} -deduce is caused by a Boolean failure (case (i) above), then \mathcal{T} -analyze_conflict produces a Boolean conflict set η^p and the corresponding value of `blevel` [ZMMM01, ZM02]; if instead the conflict is caused by a \mathcal{T} -inconsistency revealed by \mathcal{T} -solver (case (ii) or (iii) above), then the result of \mathcal{T} -analyze_conflict is the Boolean abstraction η^p of the theory conflict set $\eta \subseteq \mu$ produced by the \mathcal{T} -solver, or a mixed Boolean+theory conflict set computed by a backward-traversal of the implication graph starting from the conflicting clause $\neg\eta^p$ (see §33.4.3.5). If the \mathcal{T} -solver is not able to return a theory conflict set, the whole assignment μ may be used, after removing all Boolean literals from μ . Once the conflict set η^p and `blevel` have been computed, \mathcal{T} -backtrack behaves analogously to backtrack in DPLL: it adds the clause $\neg\eta^p$ to φ^p , either temporarily or permanently, and backtracks up to `blevel`. (These features, called \mathcal{T} -backjumping and \mathcal{T} -learning, are discussed in §33.4.3.5.)

\mathcal{T} -DPLL differs from the DPLL schema of [ZMMM01, ZM02] because it exploits:

- an extended notion of *deduction of literals*: not only *Boolean deduction* ($\mu^p \wedge \varphi^p \models_p l^p$), but also *theory deduction* ($\mu \models_{\mathcal{T}} l$);
- an extended notion of *conflict*: not only *Boolean conflicts* ($\mu^p \wedge \varphi^p \models_p \perp$), but also *theory conflicts* ($\mu \models_{\mathcal{T}} \perp$), or even *mixed Boolean+theory conflicts* ($(\mu \wedge \varphi) \models_{\mathcal{T}} \perp$). See §33.4.3.5.

Example 33.4.1. Consider the formulas φ and φ^p shown in Figure 33.4. Suppose \mathcal{T} -decide_next_branch selects, in order, $\mu^p := \{\neg B_5, B_8, B_6, \neg B_1\}$ (in c_4, c_7, c_6 , and c_1). \mathcal{T} -deduce cannot unit-propagate any literal. Assuming the enhanced version of step (iii), it invokes the \mathcal{T} -solver on $\mu := \{\neg(3x_1 - x_3 \leq 6), (x_3 = 3x_5 + 4), (x_2 - x_4 \leq 6), \neg(2x_2 - x_3 > 2)\}$. Suppose the enhanced \mathcal{T} -solver not only returns Sat, but also deduces $\neg(3x_1 - 2x_2 \leq 3)$ (c_3 and c_5) as a consequence of the first and last literals. The corresponding Boolean literal $\neg B_3$, is then added to μ^p and propagated (\mathcal{T} -propagation). As a result, A_1, A_2 and B_2 are unit-propagated from c_5, c_3 and c_2 .

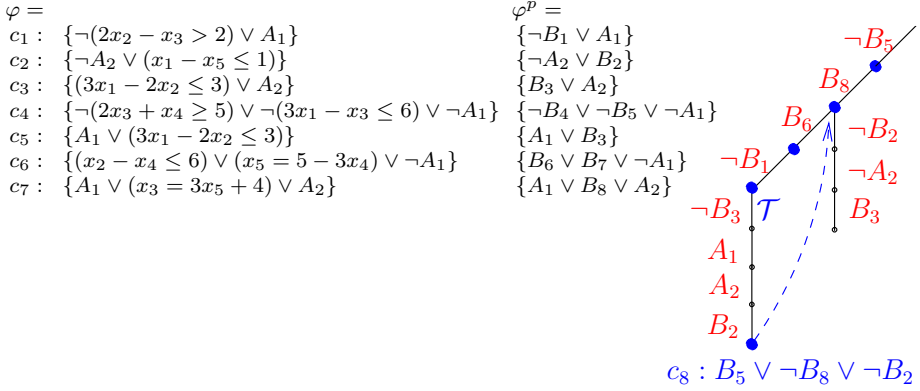


Figure 33.4. Boolean search (sub)tree in the scenario of Example 33.4.1. (A diagonal line, a vertical line and a vertical line tagged with “ \mathcal{T} ” denote literal selection, unit propagation and \mathcal{T} -propagation respectively; a bullet “ \bullet ” denotes a call to the \mathcal{T} -solver.)

Let μ^p be the resulting assignment $\{\neg B_5, B_8, B_6, \neg B_1, \neg B_3, A_1, A_2, B_2\}$. By step (iii), \mathcal{T} -deduce invokes the \mathcal{T} -solver on μ' : $\{\neg(3x_1 - x_3 \leq 6), (x_3 = 3x_5 + 4), (x_2 - x_4 \leq 6), \neg(2x_2 - x_3 > 2), \neg(3x_1 - 2x_2 \leq 3), (x_1 - x_5 \leq 1)\}$ which is inconsistent because of the 1st, 2nd, and 6th literals. As a result, the \mathcal{T} -solver returns *Unsat*, and hence \mathcal{T} -deduce returns *Conflict*. Next, \mathcal{T} -analyze_conflict and \mathcal{T} -backtrack learn the corresponding Boolean conflict clause

$$c_8 =_{def} B_5 \vee \neg B_8 \vee \neg B_2$$

and backtrack, popping from μ^p all literals up to $\{\neg B_5, B_8\}$, and then unit-propagating $\neg B_2$ on c_8 (\mathcal{T} -backjumping and \mathcal{T} -learning). Then, starting from $\{\neg B_5, B_8, \neg B_2\}$, $\neg A_2$ and B_3 are also unit-propagated (on c_2 and c_3 respectively).

As in standard DPLL, an excessive number of \mathcal{T} -learned clauses may cause an explosion in the size of φ . Thus, many lazy SMT tools introduce techniques for deleting \mathcal{T} -learned clauses when necessary. Moreover, like in standard DPLL, \mathcal{T} -DPLL can be *restarted* from scratch in order to avoid dead-end portions of the search space. The learned clauses prevent \mathcal{T} -DPLL from repeating the same steps twice. Most lazy SMT tools implement restarting mechanisms as well.

33.4.3. Enhancements to the schema

We describe some of the most effective techniques which have been proposed in order to optimize the interaction between the DPLL solver and the \mathcal{T} -solver. (We refer the reader to [Seb07] for a much more extensive and detailed survey.) Some of them derive from those developed in the context of DPLL-based procedures for modal logics (see Section 32.4.1).

33.4.3.1. Normalizing \mathcal{T} -atoms.

As discussed in Section 32.4.1, in order to avoid the generation of many trivially-unsatisfiable assignments, it is wise to preprocess \mathcal{T} -atoms so as to map as many as possible \mathcal{T} -equivalent literals into syntactically-identical ones. This can be achieved by applying some rewriting rules, like, e.g.:

- *Drop dual operators:* $(x_1 < x_2), (x_1 \geq x_2) \implies \neg(x_1 \geq x_2), (x_1 \geq x_2)$.
- *Exploit associativity:* $(x_1 + (x_2 + x_3) = 1), ((x_1 + x_2) + x_3) = 1 \implies (x_1 + x_2 + x_3 = 1)$.
- *Sort:* $(x_1 + x_2 - x_3 \leq 1), (x_2 + x_1 - 1 \leq x_3) \implies (x_1 + x_2 - x_3 \leq 1)$.
- *Exploit \mathcal{T} -specific properties:* $(x_1 \leq 3), (x_1 < 4) \implies (x_1 \leq 3)$ if x_1 represents an integer.

The applicability and effectiveness of these mappings depends on the theory \mathcal{T} .

33.4.3.2. Static learning

On some specific kinds of problems, it is possible to quickly detect *a priori* small and “obviously \mathcal{T} -inconsistent” sets of \mathcal{T} -atoms in φ (typically pairs or triplets). Some examples are:

- *incompatible values* (e.g., $\{x = 0, x = 1\}$),
- *congruence constraints* (e.g., $\{(x_1 = y_1), (x_2 = y_2), f(x_1, x_2) \neq f(y_1, y_2)\}$),
- *transitivity constraints* (e.g., $\{(x - y \leq 2), (y - z \leq 4), \neg(x - z \leq 7)\}$),
- *equivalence constraints* (e.g., $\{(x = y), (2x - 3z \leq 3), \neg(2y - 3z \leq 3)\}$).

If so, the clauses obtained by negating the literals in such sets (e.g., $\neg(x = 0) \vee \neg(x = 1)$) can be added to the formula before the search starts. Then, whenever all but one of the literals in the set are assigned to true, the negation of the remaining literal is assigned deterministically by unit propagation. This prevents the solver from generating any assignment which include the inconsistent set. This technique may significantly reduce the Boolean search space, and hence the number of calls to the \mathcal{T} -solver, producing significant speed-ups [ACG99, ACKS02, BBC⁺05a, YM06].

Intuitively, one can think of static learning as suggesting some small and “obvious” \mathcal{T} -valid lemmas relating some \mathcal{T} -atoms of φ , which drive DPLL in its Boolean search. Notice that, unlike the extra clauses added in “per-constraint” eager approaches [SSB02, SLB03] (see §33.3), the clauses added by static learning refer only to atoms which *already occur in the original formula*, so that the Boolean search space is not enlarged. Notice also that these clauses are not needed for correctness or completeness: rather, they are used only for pruning the Boolean search space.

33.4.3.3. Early pruning

Another optimization, here generically called *early pruning* – *EP*,¹⁶ is to introduce intermediate calls to the \mathcal{T} -solver while an assignment μ is still under construction (in the \mathcal{T} -DPLL scheme of §33.4.2, this corresponds to the “slightly more

¹⁶Also called *intermediate assignment checking* in [GS96] and *eager notification* in [BDS02a].

elaborate version” of step (iii) of \mathcal{T} -deduce). If \mathcal{T} -solver(μ) returns **Unsat**, then all possible extensions of μ are unsatisfiable, so \mathcal{T} -DPLL can immediately return **Unsat** and backtrack, possibly avoiding a very large amount of useless search.

In general, EP may dramatically reduce the Boolean search space, and hence of the number of calls to the \mathcal{T} -solver. Unfortunately, as EP may cause useless calls to the \mathcal{T} -solver, the benefits of the pruning effect may be partly counterbalanced by the overhead introduced by the extra calls. Many different improvements to EP and strategies for interleaving calls to the \mathcal{T} -solver with Boolean reasoning steps have been proposed [WW99, SBD02, GHN⁺04, ABC⁺02, ACGM04, NO05c, BBC⁺05b, CM06a].

33.4.3.4. \mathcal{T} -propagation

As discussed in §33.4.1, for some theories it is possible to implement the \mathcal{T} -solver so that a call to \mathcal{T} -solver(μ) returning **Sat** can also perform one or more deduction(s) of the form $\eta \models_{\mathcal{T}} l$, where $\eta \subseteq \mu$ and l is a literal on an unassigned atom in φ . If this is the case, then the \mathcal{T} -solver can return l to \mathcal{T} -DPLL, so that l^p is added to μ^p and unit-propagated [ACG99, ABC⁺02, GHN⁺04, NO05c]. This process, which is called \mathcal{T} -propagation,¹⁷ may result in new literals being assigned, leading to new calls to the \mathcal{T} -solver, followed by additional new assignments being deduced, and so on, so that together, \mathcal{T} -propagation and unit propagation may produce a much larger benefit than either of them alone. As with early-pruning, there are different strategies by which \mathcal{T} -propagation can be interleaved with unit-propagation [ACG99, ABC⁺02, GHN⁺04, BBC⁺05a, NO05c, CM06a, NOT06].

Notice that when the \mathcal{T} -solver deduces $\eta \models_{\mathcal{T}} l$, it can return this deduction to \mathcal{T} -DPLL, which can then add the \mathcal{T} -deduction clause ($\eta^p \rightarrow l^p$) to φ^p , either temporarily or permanently. The \mathcal{T} -deduction clause can be used during the rest of the search, with benefits analogous to those of \mathcal{T} -learning (see §33.4.3.5).

33.4.3.5. \mathcal{T} -backjumping and \mathcal{T} -learning

As hinted in §33.4.2, we assume that, when the \mathcal{T} -solver is invoked on a \mathcal{T} -inconsistent assignment μ , it is able to return also the conflict set $\eta \subseteq \mu$ causing the \mathcal{T} -unsatisfiability of μ (see §33.4.1). If so, \mathcal{T} -DPLL can use η^p as if it were a Boolean conflict set to drive the backjumping and learning mechanism of DPLL: the conflict clause $\neg\eta^p$ is added to φ^p either temporarily or permanently (\mathcal{T} -learning) and the procedure backtracks to the branching point suggested by η^p (\mathcal{T} -backjumping) [Hor98, PS98, WW99, dMRS02, SBD02, ABC⁺02, GHN⁺04, BBC⁺05a]. Modern implementations inherit the backjumping mechanism of current DPLL tools: \mathcal{T} -DPLL learns the conflict clause $\neg\eta^p$ and backtracks to the highest point in the stack where one $l^p \in \eta^p$ is not assigned, and unit propagates $\neg l^p$ on $\neg\eta^p$. Intuitively, DPLL backtracks to the highest point where it would have done something different if it had known in advance the conflict clause $\neg\eta^p$ from the \mathcal{T} -solver.

As hinted in §33.4.2, it is possible to use either a theory conflict set η (i.e., $\neg\eta$ is a \mathcal{T} -valid clause) or a *mixed Boolean+theory conflict set* η' , i.e., a set η'

¹⁷Also called *forward reasoning* in [ACG99], *enhanced early pruning* in [ABC⁺02], *theory propagation* in [NOT05, NO05c], and *theory-driven deduction* or \mathcal{T} -deduction in [BBC⁺05a].

s.t. an inconsistency can be derived from $\eta' \wedge \varphi$ by means of a combination of Boolean and theory reasoning ($\eta' \wedge \varphi \models_{\mathcal{T}} \perp$). Such conflict sets/clauses can be obtained starting from the theory conflict clause $\neg\eta^p$ by backward-traversal of the implication graph, until one of the standard conditions (e.g., 1UIP) is achieved. Notice that it is possible to learn *both* clauses $\neg\eta$ and $\neg\eta'$.

Example 33.4.2. The scenario depicted in Example 33.4.1 represents a form of \mathcal{T} -backjumping and \mathcal{T} -learning, in which the conflict clause c_8 is a $\mathcal{T}_{\mathcal{R}}$ -conflict clause (i.e., $\mathcal{B}2\mathcal{T}(c_8)$ is $\mathcal{T}_{\mathcal{R}}$ -valid). However, \mathcal{T} -analyze_conflict could instead look for a mixed Boolean+theory conflict clause by treating c_8 as a conflicting clause and backward-traversing the implication graph. This is done by starting with c_8 and performing resolution with each of the clauses that triggered the assignments leading to the conflict, but in the reverse order that the assignments occurred (in this case, clauses c_2 and c_3 , the antecedent clauses of B_2 and A_2 respectively, and the \mathcal{T} -deduction clause c_9 which “caused” the propagation of $\neg B_3$):

$$\begin{array}{c}
 c_8: \text{theory conflicting clause} \\
 \frac{\overbrace{B_5 \vee \neg B_8 \vee \neg B_2} \quad \overbrace{\neg A_2 \vee B_2}^{c_2}}{B_5 \vee \neg B_8 \vee \neg A_2} \quad \overbrace{B_3 \vee A_2}^{c_3}}{B_5 \vee \neg B_8 \vee B_3} \quad \overbrace{B_5 \vee B_1 \vee \neg B_3}^{c_9}}{B_5 \vee \neg B_8 \vee B_1} \\
 c'_8: \text{mixed Boolean+theory conflict clause}
 \end{array}$$

The result is the mixed Boolean+theory conflict clause $c'_8 : B_5 \vee \neg B_8 \vee B_1$. (Notice that, $\mathcal{B}2\mathcal{T}(c'_8) = (3x_1 - x_3 \leq 6) \vee \neg(x_3 = 3x_5 + 4) \vee (2x_2 - x_3 > 2)$ is not $\mathcal{T}_{\mathcal{R}}$ -valid.) If c'_8 is chosen, then \mathcal{T} -backtrack pops from μ^p all literals up to $\{\neg B_5, B_8\}$, and then unit-propagates B_1 on c'_8 , and hence A_1 on c_1 .

As with static learning, the clauses added by \mathcal{T} -learning refer only to atoms which already occur in the original formula, so that no new atom is added. [FJOS03] proposed an interesting generalization of \mathcal{T} -learning, in which learned clause may contain also new atoms. [BBC⁺05c, BBC⁺06] used a similar idea to improve the efficiency of Delayed Theory Combination (see §33.6.3). [WGG06] proposed similar ideas for a SMT tool for difference logic, in which new atoms can be generated selectively according to an ad-hoc heuristic.

33.4.3.6. Generating partial assignments

Due to the two-watched-literal scheme [MMZ⁺01], in modern implementations, DPLL returns Sat only when all variables are assigned truth values, thus returning *total* assignments. Thus, when a *partial* assignment μ is found which satisfies φ , this causes an unnecessary sequence of decisions and unit-propagations for assigning the remaining variables. In SAT, this scenario causes no extra Boolean search, because every extension of μ propositionally satisfies φ , so the overhead introduced is negligible. In SMT, however, many total assignments extending μ may be \mathcal{T} -inconsistent even though μ is \mathcal{T} -consistent, so that many useless Boolean branches and calls to \mathcal{T} -solvers may be required.

In order to overcome these problems, it is sufficient to implement some device monitoring the satisfaction of all original clauses in φ . Although this may cause some overhead in handling the Boolean component of reasoning, it may also reduce the overall Boolean search space and hence the number of subsequent calls to the \mathcal{T} -solver.

33.4.3.7. Pure-literal filtering

If we have non-Boolean \mathcal{T} -atoms occurring only positively [resp. negatively] in the input formula, we can safely drop every negative [resp. positive] occurrence of them from an assignment μ whenever μ is to be checked by the \mathcal{T} -solver [WW99, GGT01, ABC⁺02, BBC⁺05b, Seb07].¹⁸ We call this technique *pure-literal filtering*.¹⁹

There are a couple of potential benefits of this behavior. Let μ' be the filtered version of μ . First, μ' might be \mathcal{T} -satisfiable despite μ being \mathcal{T} -unsatisfiable. If so, and if μ (and hence μ') propositionally satisfies φ , then \mathcal{T} -DPLL can stop, potentially saving a lot of search. Second, if μ' (and hence μ) is \mathcal{T} -unsatisfiable, then checking the consistency of μ' rather than that of μ can be faster and result in smaller conflict sets, improving the effectiveness of \mathcal{T} -backjumping and \mathcal{T} -learning.

Moreover, this technique is particularly useful in some situations. For instance, many \mathcal{T} -solvers for $\mathcal{T}_{\mathcal{Z}}$ and its difference logic fragment cannot efficiently handle disequalities, so that they are forced to split them into a disjunction of strict inequalities. For example, the disequality $(x_1 - x_2 \neq 3)$ would be replaced by $(x_1 - x_2 > 3) \vee (x_1 - x_2 < 3)$. This causes an enlargement of the search, because the two disjuncts must be investigated separately. In many problems, however, it is common for most equalities to $(t_1 = t_2)$ occur with positive polarity only. For such equalities, pure-literal filtering avoids adding $(t_1 \neq t_2)$ to μ when $(t_1 = t_2)^p$ is assigned to false by \mathcal{T} -DPLL, so that no split is needed [ABC⁺02].

33.4.4. An abstract framework

The DPLL procedure and its variants and extensions, including \mathcal{T} -DPLL, can also be described more abstractly as transition systems. This allows one to ignore unimportant control and implementation details and provide the essence of each variant in terms of a set of state transition rules and a rule application strategy.

Following the *Abstract DPLL Modulo Theories* framework first introduced in [NOT05], the variants of \mathcal{T} -DPLL discussed in the previous subsection can be described abstractly as a transition relation over states of the form Fail or $\mu \parallel \varphi$, where φ is a (ground) CNF formula, or, equivalently, a finite set of clauses, and μ is a *sequence* of (ground) literals, each marked as a *decision* or a non-decision literal. As in §33.4.1, the set μ represents a partial assignment of truth values to the atoms of φ . The transition relation is specified by a set of *transition rules*, given below. In the rules, we denote the concatenation of sequences of literals

¹⁸If both \mathcal{T} -propagation and pure-literal filtering are implemented, then the filtered literals must be dropped not only from the assignment, but also from the list of literals which can be \mathcal{T} -deduced, so that to avoid the \mathcal{T} -propagation of literals which have been filtered away.

¹⁹Also called *triggering* in [WW99, ABC⁺02].

by simple juxtaposition (e.g., $\mu \mu' \mu''$), treating single literals as one element sequences and denoting the empty sequence with \emptyset . To denote that a literal l is annotated as a decision literal in a sequence we write it as l^\bullet . A literal l is *undefined* in μ if neither l nor $\neg l$ occurs in μ . We write $S \Longrightarrow S'$ as usual to mean that two states S and S' are related by the transition relation \Longrightarrow and say that there is a *transition* from S to S' . We call any sequence of transitions of the form $S_0 \Longrightarrow S_1 \Longrightarrow S_2 \Longrightarrow \dots$ a *derivation*.

Definition 33.4.3 (Transition Rules). The following is a set of rules for *Abstract \mathcal{T} -DPLL*. Except for **Decide**, all the rules that introduce new literals annotate them as non-decision literals.

$$\text{Propagate : } \mu \parallel \varphi, c \vee l \Longrightarrow \mu l \parallel \varphi, c \vee l \text{ if } \begin{cases} \mu \models_p \neg c \\ l \text{ is undefined in } \mu \end{cases}$$

$$\text{Decide : } \mu \parallel \varphi \Longrightarrow \mu l^\bullet \parallel \varphi \text{ if } \begin{cases} l \text{ or } \neg l \text{ occurs in } \varphi \\ l \text{ is undefined in } \mu \end{cases}$$

$$\text{Fail : } \mu \parallel \varphi, c \Longrightarrow \text{Fail} \text{ if } \begin{cases} \mu \models_p \neg c \\ \mu \text{ contains no decision literals} \end{cases}$$

$$\text{Restart : } \mu \parallel \varphi \Longrightarrow \emptyset \parallel \varphi$$

$$\mathcal{T}\text{-Propagate : } \mu \parallel \varphi \Longrightarrow \mu l \parallel \varphi \text{ if } \begin{cases} \mu \models_{\mathcal{T}} l \\ l \text{ or } \neg l \text{ occurs in } \varphi \\ l \text{ is undefined in } \mu \end{cases}$$

$$\mathcal{T}\text{-Learn : } \mu \parallel \varphi \Longrightarrow \mu \parallel \varphi, c \text{ if } \begin{cases} \text{each atom of } c \text{ occurs in } \mu \parallel \varphi \\ \varphi \models_{\mathcal{T}} c \end{cases}$$

$$\mathcal{T}\text{-Forget : } \mu \parallel \varphi, c \Longrightarrow \mu \parallel \varphi \text{ if } \{ \varphi \models_{\mathcal{T}} c \}$$

\mathcal{T} -Backjump :

$$\mu l^\bullet \mu' \parallel \varphi, c \Longrightarrow \mu k \parallel \varphi, c \text{ if } \begin{cases} \mu l^\bullet \mu' \models_p \neg c, \text{ and there is} \\ \text{some clause } c' \vee l' \text{ such that:} \\ \varphi, c \models_{\mathcal{T}} c' \vee l' \text{ and } \mu \models_p \neg c', \\ l' \text{ is undefined in } \mu, \text{ and} \\ l \text{ or } \neg l \text{ occurs in } \mu l^\bullet \mu' \parallel \varphi \end{cases}$$

The clauses c and $c' \vee l'$ in the \mathcal{T} -Backjump rule are respectively the *conflicting* clause and the *backjump* clause of the rule.

The rules **Propagate**, **Decide**, **Fail** and **Restart**, operate at the propositional level. The other rules involve the theory \mathcal{T} and have rather general preconditions. While all of these preconditions are decidable whenever the \mathcal{T} -satisfiability of sets of ground literals is decidable, they might be expensive to check in their full generality.²⁰ However, there exist restricted applications of these rules that are both efficient and enough for completeness [NOT06]. Given a ground CNF formula φ , the purpose of the rules above is to extend and modify an originally empty sequence until it determines a total, satisfying assignment for φ or the **Fail** rule becomes applicable.

²⁰In particular, the precondition of \mathcal{T} -Backjump seems problematic on a first look because it relies on the computability of the backjump clause $c' \vee l'$.

Example 33.4.4. The computation discussed in Example 33.4.2 can be described by the following derivation in Abstract \mathcal{T} -DPLL, where φ is again the formula consisting of the $\mathcal{T}_{\mathcal{R}}$ -clauses c_1, \dots, c_7 in Figure 33.4 and c_8 is the clause abstracted by $B_5 \vee \neg B_8 \vee \neg B_2$. For space constraints, instead of φ 's literals we use their propositional abstraction here, and use \Longrightarrow^+ to denote multiple transitions.

1–4.	$\emptyset \parallel \varphi \Longrightarrow^+$	$\neg B_5 \bullet B_8 \bullet B_6 \bullet \neg B_1 \bullet \parallel \varphi$	(by Decide)
5.	\Longrightarrow	$\neg B_5 \bullet B_8 \bullet B_6 \bullet \neg B_1 \bullet \neg B_3 \parallel \varphi$	(by \mathcal{T} -Propagate)
6–8.	\Longrightarrow^+	$\neg B_5 \bullet B_8 \bullet B_6 \bullet \neg B_1 \bullet \neg B_3 A_1 A_2 B_2 \parallel \varphi$	(by Propagate)
9.	\Longrightarrow	$\neg B_5 \bullet B_8 \bullet B_6 \bullet \neg B_1 \bullet \neg B_3 A_1 A_2 B_2 \parallel \varphi, c_8$	(by \mathcal{T} -Learn)
10.	\Longrightarrow	$\neg B_5 \bullet B_8 \bullet B_1 \parallel \varphi, c_8$	(by \mathcal{T} -Backjump)
11.	\Longrightarrow	$\neg B_5 \bullet B_8 \bullet B_1 A_1 \parallel \varphi, c_8$	(by Propagate)

Recall that clause c_8 in Step 9 is a theory lemma added in response to the \mathcal{T} -inconsistency of the assignment produced by Step 8. In step 10, \mathcal{T} -Backjump is applied with conflicting clause $c = c_8$ and backjump clause $c' \vee l' = B_5 \vee \neg B_8 \vee B_1$ with $l' = B_1$. The backjump clause is derived as explained in Example 33.4.2.

Let us say that a state is \mathcal{T} -compatible if it is *Fail* or has the form $\mu \parallel \varphi$ where μ is \mathcal{T} -consistent. The transition rules can be used to decide the \mathcal{T} -satisfiability of an input formula φ_0 by generating a derivation

$$\emptyset \parallel \varphi_0 \Longrightarrow_{\mathcal{B}} S_1 \Longrightarrow_{\mathcal{B}} \dots \Longrightarrow_{\mathcal{B}} S_n, \quad (33.1)$$

where S_n is a \mathcal{T} -compatible state to which none of the rules *Propagate*, *Decide*, *Fail*, or *\mathcal{T} -Backjump* applies. We call such a derivation *exhausted*.

To generate from $\emptyset \parallel \varphi_0$ only derivations like (33.1) above it is enough to impose a few, rather weak, restrictions on a rule application strategy. Roughly speaking, these restrictions rule out only derivations with subderivations consisting exclusively of \mathcal{T} -Learn and \mathcal{T} -Forget steps, and derivations that do not apply *Restart* with increased periodicity.²¹

A rule application strategy is *fair* if it conforms to these restrictions and stops (extending) a derivation only when the derivation is exhausted. Every fair strategy is *terminating*, *sound*, and *complete* in the following sense (see [NOT06] for details).

Termination: Starting from a state $\emptyset \parallel \varphi_0$, the strategy generates only finite derivations.

Soundness: If φ_0 is \mathcal{T} -satisfiable, every exhausted derivation of $\emptyset \parallel \varphi_0$ generated by the strategy ends with a state of the form $\mu \parallel \varphi$ where μ is a (\mathcal{T} -consistent) total, satisfying assignment for φ .

Completeness: If φ_0 is not \mathcal{T} -satisfiable, every exhausted derivation of $\emptyset \parallel \varphi_0$ generated by the strategy ends with *Fail*.

In the setting above, fair strategies stop a derivation for a \mathcal{T} -satisfiable φ_0 only once they compute a *total* assignment for φ_0 's atoms. A more general setting can be obtained, with similar results, by defining a finite derivation to be exhausted if its last state is \mathcal{T} -compatible and, when the state is $\mu \parallel \varphi$, the assignment μ

²¹In fact, the actual restrictions are even weaker. See Theorem 3.7 of [NOT06] for details.

propositionally satisfies φ . We refer back to the discussion in §33.4.3.6 for why this can be more convenient computationally.

The rule set in Definition 33.4.3 is not minimal. For instance, there are fair strategies that use only \mathcal{T} -Propagate or only \mathcal{T} -Learn in addition to Decide, Fail, Propagate and \mathcal{T} -Backjump. The rule set is not exhaustive either because it models only the most common (and experimentally most useful) operations found in lazy SMT solvers. We explain how below, focusing on the less trivial rules.

Decide. This rule represents a case split by adding an undefined literal of φ , or its negation, to μ . The added literal l is annotated as a *decision literal*, to denote that if μl cannot be extended to a model of φ then an alternative extension of μ must be considered—something done with the \mathcal{T} -Backjump rule.²²

\mathcal{T} -Propagate. This rule performs the kind of theory propagation described in §33.4.3.4 by adding to the current assignment μ any undefined literal l that is \mathcal{T} -entailed by μ . The rule's precondition maintains the invariant that every atom in an assignment occurs in the initial formula of the derivation. This invariant is crucial for termination.

\mathcal{T} -Backjump. This rule models the kind of *conflict-driven* backjumping described in §33.4.3.5. As defined, the rule is only triggered by a propositional conflict with one of the current clauses ($\mu \models_p \neg c$). This is for simplicity and uniformity, and without loss of generality thanks to the \mathcal{T} -Propagate and \mathcal{T} -Learn rules. Using those rules, any conflict involving the theory \mathcal{T} is reducible to a propositional conflict. For instance, if the current assignment μ has a \mathcal{T} -inconsistent subset η , then $\emptyset \models_{\mathcal{T}} \neg\eta$. The theory lemma $\neg\eta$ can then be added to the current clause set φ by one application of \mathcal{T} -Learn and then play the role of clause c in \mathcal{T} -Backjump.

\mathcal{T} -Backjump is triggered by the existence of a conflicting clause, but in order to undo (the effect of) an earlier decision step it needs a backjump clause $c' \vee l'$, which synthesizes the reason of the conflict, indicating the level to backjump to and the literal to put in place of the decision literal for that level. This clause is the dual of the conflict set η discussed in §33.4.3.5; that is, $c' \vee l' = \neg\eta$.

\mathcal{T} -Learn. This rule allows the addition to the current formula φ of an arbitrary clause c that is \mathcal{T} -entailed by φ and consists of atoms in the current state. It can be used to model the static learning techniques described in §33.4.3.2 as well as the usual conflict-driven lemma learning that adds backjump clauses, or any other technique that takes advantage of theory consequences of φ . In particular, it can be used to model uniformly all the early pruning techniques described in §33.4.3.3. This is because any backjumping motivated by the \mathcal{T} -inconsistency of the current assignment μ can be modeled as the discovery and learning of a theory lemma c that is propositionally falsified by μ , followed by an application of \mathcal{T} -Backjump with c as the conflicting clause. The rule does not model the learning of clauses containing *new* literals, which is done by some SMT solvers, as seen in

²²Note that the precondition of Decide does not include a check on whether Propagate, say, applies to l or its negation. This is intentional since such control considerations are best left to a rule application strategy.

§33.4.3.5. An extension of Abstract \mathcal{T} -DPLL that allows that is discussed later in §33.5.2.

We can now describe some of the approaches for implementing \mathcal{T} -DPLL solvers discussed in §33.4.2 in terms of Abstract \mathcal{T} -DPLL. (See [NOT06] for more details.)

Offline integrations of a theory solver and a DPLL solver are modeled by a class of rule application strategies that do not use the \mathcal{T} -Propagate rule. Whenever a strategy in this class produces a state $\mu \parallel \varphi$ irreducible by Decide, Fail, Propagate, or \mathcal{T} -Backjump, the sequence μ is a satisfying assignment for the formula φ_0 in the initial state $\emptyset \parallel \varphi_0$, but it may not be \mathcal{T} -consistent. If it is not, there exists a $\eta \subseteq \mu$ such that $\emptyset \models_{\mathcal{T}} \neg\eta$. The strategy then adds the theory lemma $\neg\eta$ (the blocking clause) with one \mathcal{T} -Learn step and applies Restart, repeating the same cycle above until a \mathcal{T} -compatible state is generated. With an incremental \mathcal{T} -solver it might be more convenient to check the \mathcal{T} -consistency of μ in a state $\mu \parallel \varphi$ even if the state is reducible by one of the four rules above. In that case, it is possible to learn a blocking clause and restart earlier. Strategies like these are sound and complete. To be fair, and so terminating, they must not remove (with \mathcal{T} -Forget) any blocking clause.

A strategy can take advantage of an online SAT-solver by preferring backjumping to systematic restarting after μ is found \mathcal{T} -inconsistent. This is done by first learning a lemma $\neg\eta$ for some $\eta \subseteq \mu$, and then *repairing* μ using that lemma. In fact, since $\neg\eta$ is a conflicting clause by construction, either \mathcal{T} -Backjump or Fail applies—depending respectively on whether μ contains decision literals or not. To be fair, strategies that apply \mathcal{T} -Backjump/Fail instead of Restart do not need to keep the blocking clause once they use it as the conflicting clause for backjumping.

Any fair strategy above remains fair if it is modified to interleave arbitrary applications of \mathcal{T} -Propagate. Stronger results are possible if \mathcal{T} -Propagate is applied eagerly (in concrete, if it has higher priority than Decide). In that case, it is impossible to generate \mathcal{T} -inconsistent assignments, and so it is unnecessary for correctness to learn any blocking clauses, or any theory lemmas at all.

33.5. Theory Solvers

The lazy approach to SMT as described above relies on combining a SAT solver with a *theory solver* for some theory \mathcal{T} . The role of the theory solver is to accept a set of literals and report whether the set is \mathcal{T} -satisfiable or not. Typically, theory solvers are *ad hoc*, with specialized decision procedure tailored to the theory in question. For details on such procedures for some common theories, we refer the reader to the references given in §33.2.2. Because different theories often share some characteristics, a natural question is whether there exist general or parameterized methods that have broader applicability. This section discusses several such approaches that have been developed: Shostak's method (§33.5.1), splitting on demand (§33.5.2), layered theory solvers (§33.5.3), and rewriting-based theory solvers (§33.5.4).

33.5.1. Shostak's method

Shostak's method was introduced in a 1984 paper [Sho84] as a general method for combining the theory of equality with one or more additional theories that satisfy certain specific criteria. The original paper lacks rigor and contains serious errors, but work since then has shed considerable light on the original claims [CLS96, RS01, SR02, KC03, BDS02b, Gan02]. We summarize some of the most significant results here. We start with some definitions.

Definition 33.5.1. A set \mathcal{S} of equations is said to be in *solved form* iff the left-hand side of each equation in \mathcal{S} is a ground constant which appears only once in \mathcal{S} . We will refer to these constants appearing only on the left-hand sides as *solitary* constants.

A set \mathcal{S} of equations in solved form defines an idempotent substitution: the one which replaces each solitary constant with its corresponding right-hand side. If S is a term or set of terms, we denote the result of applying this substitution to S by $\mathcal{S}(S)$.

Definition 33.5.2. Given a formula F and a formula G , we define $\gamma_F(G)$ as follows:

1. Let G' be the formula obtained by replacing each free constant symbol in G that does not appear in F with a fresh variable.
2. Let \bar{v} be the set of all fresh variables introduced in the previous step.
3. Then, $\gamma_F(G) = \exists \bar{v}. G'$.

Definition 33.5.3. A consistent theory \mathcal{T} with signature Σ is a *Shostak* theory if the following conditions hold.

1. Σ does not contain any predicate symbols.
2. There exists a *canonizer* *canon*, a computable function from Σ -terms to Σ -terms, with the property that $\models_{\mathcal{T}} s = t$ iff $\text{canon}(s) \equiv \text{canon}(t)$.
3. There exists a *solver* *solve*, a computable function from Σ -equations to sets of formulas defined as follows:
 - (a) If $\models_{\mathcal{T}} s \neq t$, then $\text{solve}(s = t) \equiv \{\perp\}$.
 - (b) Otherwise, $\text{solve}(s = t)$ returns a set \mathcal{S} of equations in solved form such that $\models_{\mathcal{T}} (s = t) \leftrightarrow \gamma_{s=t}(\mathcal{S})$.

The main result in Shostak's paper is that given a Shostak theory \mathcal{T} , a simple algorithm can be used to determine the satisfiability of conjunctions of Σ -literals. Algorithm *S1* (shown in Figure 33.5) makes use of the properties of a Shostak theory to check the joint satisfiability of an arbitrary set of equalities, Γ , and an arbitrary set of disequalities, Δ , in a Shostak theory with canonizer *canon* and solver *solve*.

Algorithm *S1* is sound and complete whenever Δ contains at most one disequality or if \mathcal{T} satisfies the additional requirement of being convex (as defined in §33.2.1.2).

```

S1( $\Gamma$ ,  $\Delta$ , canon, solve)
1.  $\mathcal{S} := \emptyset$ ;
2. WHILE  $\Gamma \neq \emptyset$  DO BEGIN
3.   Remove some equality  $s = t$  from  $\Gamma$ ;
4.    $s^* := \mathcal{S}(s)$ ;  $t^* := \mathcal{S}(t)$ ;
5.    $\mathcal{S}^* := \text{solve}(s^* = t^*)$ ;
6.   IF  $\mathcal{S}^* = \{ \perp \}$  THEN RETURN FALSE;
7.    $\mathcal{S} := \mathcal{S}^*(\mathcal{S}) \cup \mathcal{S}^*$ ;
8. END
9. IF canon( $\mathcal{S}(s)$ )  $\equiv$  canon( $\mathcal{S}(t)$ ) for some  $s \neq t \in \Delta$  THEN RETURN FALSE;
10. RETURN TRUE;

```

Figure 33.5. Algorithm S1: A simple satisfiability checker based on Shostak's algorithm

Example 33.5.4. Perhaps the most obvious example of a Shostak theory is $T'_{\mathcal{R}}$. A simple canonizer for this theory can be obtained by imposing an order on all ground constants and combining like terms. For example, $\text{canon}(c + 3b - a - 5c) \equiv -a + 3b + (-4c)$. Similarly, a solver can be obtained simply by solving for one of the constants in an equation (returning \perp if no solution exists). For this theory, Algorithm *S1* corresponds to Gaussian elimination with back-substitution. Consider the following set of literals: $\{a + 3b - 2c = 1, a - b - 6c = 1, b + a \neq a - c\}$. The following table shows the values of Γ , \mathcal{S} , $s^* = t^*$, and \mathcal{S}^* on each iteration of Algorithm *S1* starting with $\Gamma = \{a + 3b - 2c = 1, a - b - 6c = 1\}$:

Γ	\mathcal{S}	$s^* = t^*$	\mathcal{S}^*
$a + 3b - 2c = 1$ $a - b - 6c = 1$	\emptyset	$a + 3b - 2c = 1$	$a = 1 - 3b + 2c$
$a - b - 6c = 1$	$a = 1 - 3b + 2c$	$1 - 3b + 2c - b - 6c = 1$	$b = -c$
\emptyset	$a = 1 + 5c$ $b = -c$		

Now, notice that $\text{canon}(\mathcal{S}(b + a)) \equiv \text{canon}(-c + 1 + 5c) \equiv 1 + 4c$ and $\text{canon}(\mathcal{S}(a - c)) \equiv \text{canon}(1 + 5c - c) \equiv 1 + 4c$. Since $b + a \neq a - c \in \Delta$, the algorithm returns FALSE indicating that the original set of literals is unsatisfiable in \mathcal{T} .

33.5.1.1. Combining Shostak theories

Besides providing a satisfiability procedure for a single Shostak theory \mathcal{T} , the original paper makes several additional claims. The first is that a variation of Algorithm *S1* can be used to decide the satisfiability of any theory \mathcal{T}' , where \mathcal{T}' is the extension of \mathcal{T} after adding an arbitrary number of constant and function symbols. In other words, there is an algorithm for the *combination* of \mathcal{T} with the theory of equality. This claim has been clarified and proved to be correct in later work [RS01, Gan02], and we do not elaborate on it here.

The second claim regarding combinations of theories is that given any two Shostak theories, their canonizers and solvers can be combined to obtain a decision procedure for the combined theories. While it is true that canonizers can be

combined (see [KC03, SR02]), it was shown in [KC03] that solvers can almost never be combined, and thus Shostak’s method as originally presented does not provide a way to combine theories (beyond simple combinations of a single Shostak theory with the theory of equality). In [SR02], a correct method for combining Shostak theories is given. However, the method does not combine theory solvers as proposed by Shostak, but relies, instead, on the Nelson-Oppen framework covered in §33.6.1.

33.5.2. Splitting on demand

Thus far, we have assumed that a theory solver \mathcal{T} -solver for a theory \mathcal{T} takes as input a set of literals and outputs true if the set is \mathcal{T} -consistent and false otherwise. For some important theories, determining the \mathcal{T} -consistency of a conjunction of literals requires *internal* case splitting (i.e. case splitting within the \mathcal{T} -solver).

Example 33.5.5. In the theory $\mathcal{T}_{\mathcal{A}}$ of arrays introduced in §33.2.2, consider the following set of literals: $read(write(A, i, v), j) = x, read(A, j) = y, x \neq v, x \neq y$. To see that this set is unsatisfiable, notice that if $i = j$, then $x = v$ because the value read should match the value written in the first equation. On the other hand, if $i \neq j$, then $x = read(A, j)$ and thus $x = y$. Deciding the $\mathcal{T}_{\mathcal{A}}$ -consistency of larger sets of literals may require a significant amount of such reasoning by cases.

Because theories like $\mathcal{T}_{\mathcal{A}}$ require internal case splits, solving problems with general Boolean structure over such theories using the framework developed in §33.4 results in a system where case splitting occurs in two places: in the Boolean DPLL (SAT) engine as well as inside the theory solver. In order to simplify the implementation of theory solvers for such theories, and to centralize the case splitting in a single part of the system, it is desirable to allow a theory solver \mathcal{T} -solver to *demand* that the DPLL engine do additional case splits before determining the \mathcal{T} -consistency of a partial assignment. For flexibility—and because it is needed by actual theories of interest—the theory solver should be able to demand case splits on literals that may be unfamiliar to the DPLL engine and may possibly even contain fresh constant symbols. Here, we give a brief explanation of how this can be done in the context of the abstract framework given in §33.4.4. Details can be found in [BNOT06a].

Recall that in the abstract framework, the \mathcal{T} -Learn rule allows the theory solver to add an arbitrary clause to those being processed by the DPLL engine, so long as all the atoms in that clause are already known to the DPLL engine. Our approach will be to relax this restriction. It is not hard to see, however, that this poses a potential termination problem. We can overcome this difficulty so long as for any input formula ϕ , the set of all literals needed to check the \mathcal{T} -consistency of ϕ is finite. We formalize this notion by introducing the following definition.

Definition 33.5.6. \mathcal{L} is a *suitable literal-generating function* if for every finite set of literals L :

1. \mathcal{L} maps L to a new finite set of literals L' such that $L \subseteq L'$.
2. For each atomic formula α , $\alpha \in \mathcal{L}(L)$ iff $\neg\alpha \in \mathcal{L}(L)$.

3. If L' is a set of literals and $L \subseteq L'$, then, $\mathcal{L}(L) \subseteq \mathcal{L}(L')$ (monotonicity).
4. $\mathcal{L}(\mathcal{L}(L)) = \mathcal{L}(L)$ (idempotence).

For convenience, given a formula ϕ , we denote by $\mathcal{L}(\phi)$ the result of applying \mathcal{L} to the set of all literals appearing in ϕ . In order to be able to safely use splitting on demand for a theory solver \mathcal{T} -solver, we must be able to show the existence of a suitable literal-generating function \mathcal{L} such that: for every input formula ϕ , the set of all literals on which the \mathcal{T} -solver may demand case splits when starting with a conjunction of literals from ϕ is contained in $\mathcal{L}(\phi)$. For example, for \mathcal{T}_A , $\mathcal{L}(\phi)$ could contain atoms of the form $i = j$, where i and j are array indices occurring in ϕ . Note that there is no need to explicitly construct $\mathcal{L}(\phi)$. It is enough to know that it exists.

As mentioned above, it is sometimes useful to demand case splits on literals containing new constant symbols. The introduction of new constant symbols poses potential problems not only for termination, but also for soundness. This is because the abstract framework relies on the fact that whenever $\emptyset \parallel \phi$ reduces in one or more steps to $\mu \parallel \phi'$, the formulas ϕ and ϕ' are \mathcal{T} -equivalent. This is no longer true if we allow the introduction of new constant symbols. Fortunately, it is sufficient to ensure \mathcal{T} -equisatisfiability of ϕ and ϕ' . With this in mind, we can give a new transition rule called **Extended \mathcal{T} -Learn** which replaces \mathcal{T} -Learn and allows for the desired additional flexibility.

Definition 33.5.7. The *Extended DPLL Modulo Theories system*, consists of the rules of §33.4.4 except that the \mathcal{T} -Learn rule is replaced by the following²³ rule:

Extended \mathcal{T} -Learn

$$\mu \parallel \phi \quad \Longrightarrow \quad \mu \parallel \phi, c \quad \mathbf{if} \quad \left\{ \begin{array}{l} \text{each atom of } c \text{ occurs in } \phi \text{ or in } \mathcal{L}(\mu) \\ \phi \models_{\mathcal{T}} \gamma_{\phi}(c) \end{array} \right.$$

The key observation is that an implementation using **Extended \mathcal{T} -Learn** has more flexibility when a state $\mu \parallel \phi$ is reached that is final with respect to the basic rules **Propagate**, **Decide**, **Fail**, and **\mathcal{T} -Backjump**. Whereas before it would have been necessary to determine the \mathcal{T} -consistency of μ when such a state was reached, the **Extended \mathcal{T} -Learn** rule allows the possibility of *delaying* a response by demanding that additional case splits be done first. As shown in [BNOT06a], the extended framework retains soundness and completeness. Furthermore, the properties of \mathcal{L} ensure that a delayed response cannot be delayed indefinitely, and thus the framework also ensures termination under similar conditions as the original framework.

Example 33.5.8. A careful examination of the decision procedure for \mathcal{T}_A given in [SDBL01] reveals the following:

1. Each term can be categorized as an *array* term, an *index* term, a *value* term, or a *set* term.
2. No new array terms are ever introduced by the inference rules.
3. At most one new index term for every pair of array terms is introduced.
4. Set terms are made up of some finite number of index terms.
5. The only new value terms introduced are of the form $read(a, i)$ where a is an array term and i is an index term.

²³The definition of γ was given in §33.5.1.

It follows that the total number of possible terms that can be generated by the procedure starting with any finite set of literals is finite. Because there are only a finite number of predicates, it then follows that this set of rules is literal-bounded.

In general, a similar analysis must be done for every theory before it can be integrated with the Extended DPLL Modulo Theories framework as described above. It should also be pointed out that the extended framework requires a DPLL engine that is capable of dealing with a dynamically expanding set of literals. However, because the splitting on demand approach can result in drastically simpler theory solvers, it remains an attractive and useful implementation strategy.

The approach can be refined to work with several theory solvers when the background theory \mathcal{T} is a combination of the solver's theories [BNOT06b, BNOT06a]. Then, the lemma generation mechanism is also used to achieve a sound and complete cooperation among the theory solvers, in the spirit of the Nelson-Oppen combination method described in §33.6. In this form, splitting on demand is implemented in the CVC, CVC Lite and CVC3 systems [SBD02, BB04, BT07]. A major requirement of the refinement is that each theory solver must be aware that it is being combined with others so that it can generate suitable lemmas necessary for the cooperation. An alternative approach to achieving inter-solver cooperation through the DPLL engine that does not have this requirement is described in some detail in §33.6.3.

33.5.3. Layered theory solvers

Sometimes, a theory \mathcal{T} has the property that a fully general solver for \mathcal{T} is not always needed: rather, the unsatisfiability of an assignment μ can often be established in less expressive, but much easier, sub-theories. Thus, the \mathcal{T} -solver may be organized in a *layered hierarchy* of solvers of increasing solving capabilities [ABC⁺02, BBC⁺05a, SS05, CM06b, BCF⁺07]. The general idea consists of stratifying the problem over N layers L_0, L_1, \dots, L_{N-1} of increasing complexity, and searching for a solution “at as simple a level as possible”. If one of the simpler solvers finds a conflict, then this conflict is used to prune the search at the Boolean level; if it does not, the next solver in the sequence is activated.

Since L_{n+1} refines L_n , if the set of literals is not satisfiable at level L_n , then it is not at L_{n+1}, \dots, L_{N-1} . If indeed a model S exists at L_n , either n equals $N-1$, in which case S solves the problem, or a refinement of S must be searched for at L_{n+1} . In this way, much of the reasoning can be performed at a high level of abstraction. This results in increased efficiency during the search for a solution, since the later solvers, which are often responsible for most of the complexity, are avoided whenever possible.

The schema can be further enhanced by allowing each layer L_i to infer novel equalities and inequalities and to pass them down to the next layer L_{i+1} , so as to better drive its search [SS05, SS06, CM06b].

33.5.4. Rewriting-based theory solvers

Another approach for building theory solvers relies on the power and flexibility of modern automated theorem provers, in particular, provers based on the *superposition calculus* [NR01], a modern version of the resolution calculus for first-order

logic with equality. This calculus is based on term rewriting techniques and comes equipped with powerful redundancy criteria that allow one to build very effective control strategies for reducing the search space.

The superposition-based approach to SMT, first proposed in [ARR03] and then further elaborated upon in [ABRS05, BGN⁺06a, BE07a, BE07b], applies to theories \mathcal{T} that are axiomatizable by a finite (and relatively small) set of first-order clauses, such as for instance the theory of arrays in §33.2.2. The main idea is to instrument a superposition prover with specialized control strategies which, together with the axioms of \mathcal{T} , effectively turn the prover into a decision procedure for ground \mathcal{T} -satisfiability.

An advantage of the approach is a simplified proof of correctness, even in the case of combined theories, which reduces to a routine proof of termination of the application of the various superposition rules (see, e.g., [ARR03] for details). Another potential advantage is the reuse of efficient data structures and algorithms for automated deduction implemented in state-of-the-art theorem provers. The main disadvantage is that to get additional features typically required in SMT such as model generation, incrementality, and so on, one may need to modify the theorem prover in ways not foreseen by the original implementors, possibly at the cost of a considerable (re)implementation effort. While this approach has generated an active stream of interesting theoretical work, its practical impact has been limited so far by a scarcity of robust and competitive implementations.

33.6. Combining Theories

We mentioned that in SMT one is often faced with formulas involving several theories at once. This is particularly true in software verification, where proof obligations are formulas talking about several datatypes, each modeled by its own theory. We have seen that for many of these theories the ground satisfiability problem is decidable, and often by efficient theory solvers. Hence, a natural question is whether it is possible to combine theory solvers for several *component* theories $\mathcal{T}_1, \dots, \mathcal{T}_n$ *modularly* into a theory solver that decides ground satisfiability modulo their combination $\mathcal{T}_1 \oplus \dots \oplus \mathcal{T}_n$.

In general, the answer is negative, simply because there exist theories with a decidable ground satisfiability problem whose combination has an undecidable ground satisfiability problem (see, e.g., [BGN⁺06b]). Positive results are however possible by imposing restrictions on the component theories and their combination. A successful combination method for theory solvers is due to Nelson and Oppen [NO79]. The success of the method is based on the fact that its restrictions on the theories are not very strong in practice, and lead to efficient implementations. It is fair to say that most work in theory combination in SMT is based on extensions and refinements of Nelson and Oppen's work (see, e.g., [TH96, Rin96, BDS02c, TR03, TZ04, Ghi04, TZ05, RRZ05, GNZ05, BBC⁺06, GNRZ06, CK06, BNOT06a, KGGT07]).

In this section, we present a declarative non-deterministic combination framework, first presented in [Opp80b], that captures the essence of the original combination procedure by Nelson and Oppen in [NO79], and briefly discuss a few

variants and extensions. Then we describe how an efficient implementation of this framework can be incorporated into the lazy approach to SMT.

For simplicity, we consider just the case of combining two theories and their solvers since the case of more theories is analogous. Therefore, let Σ_1, Σ_2 be two signatures and let \mathcal{T}_i be a Σ_i theory for $i = 1, 2$. The combination of \mathcal{T}_1 and \mathcal{T}_2 will be the theory $\mathcal{T}_1 \oplus \mathcal{T}_2$ as defined in §33.2.1.3.

33.6.1. A Logical Framework for Nelson-Oppen Combination

The original Nelson-Oppen method applies to theories \mathcal{T}_1 and \mathcal{T}_2 with disjoint signatures and each equipped with a theory solver deciding a ground \mathcal{T}_i -satisfiability problem. The gist of the method is to take a ground formula of signature $\Sigma_1 \cup \Sigma_2 \cup C$, where C is a set of constant symbols not in $\Sigma_1 \cup \Sigma_2$, convert it into an equisatisfiable conjunction $\varphi_1 \wedge \varphi_2$ with each φ_i of signature $\Sigma_i \cup C$, and feed each φ_i to \mathcal{T}_i 's theory solver. The two solvers cooperate by exchanging entailed constraints about their respective formulas until one of them has enough information to decide the satisfiability in $\mathcal{T}_1 \oplus \mathcal{T}_2$ of the original formula.

Any version of the Nelson-Oppen method is more conveniently described by considering only $(\Sigma_1 \cup \Sigma_2 \cup C)$ -formulas φ that are just *conjunctions of literals*. This restriction is without loss of generality both in theory, via a preliminary conversion to disjunctive normal form, and in practice, as we will see in §33.6.3. Now, queries like φ cannot be processed directly by either theory solver unless they have the *pure form* $\varphi_1 \wedge \varphi_2$ where each φ_i is a $(\Sigma_i \cup C)$ -formula—possibly \top . Even then, however, using the theory solvers in isolation is not enough because it is possible for $\varphi_1 \wedge \varphi_2$ to be $\mathcal{T}_1 \oplus \mathcal{T}_2$ -unsatisfiable while each φ_i is \mathcal{T}_i -satisfiable. By a simple application of Craig's interpolation lemma [Cra57] this situation happens precisely when there is a first-order formula ψ over the signature C such that $\varphi_1 \models_{\mathcal{T}_1} \psi$ and $\varphi_2 \wedge \psi \models_{\mathcal{T}_2} \perp$. Note that Craig's lemma tells us here that $=$ is the only predicate symbol and the elements of C are the only function symbols occurring in ψ . But it does not provide any indication of whether ψ has quantifiers or not, and which ones. One of the main theoretical contributions of Nelson and Oppen's work was to show that, under the right conditions, ψ is actually ground. Moreover, it is also computable.

The Nelson-Oppen method is not limited to formulas in pure form because any ground formula can be (efficiently) turned into an equisatisfiable pure form by a suitable *purification procedure*. The procedure most commonly used (and its correctness proof) is straightforward but to describe it we need some definitions and notation first.

Let $\Sigma = \Sigma_1 \cup \Sigma_2 \cup C$ and fix $i \in \{1, 2\}$. A Σ -term t is an *i-term* if its top function symbol is in $\Sigma_i \cup C$. A Σ -literal α is an *i-literal* if its predicate symbol is in $\Sigma_i \cup C$ or if it is of the form $(\neg)s = t$ and both s and t are *i-terms* or $i = 1$.²⁴ A subterm of an *i-atom* α is an *alien* subterm of α if it is a *j-term*, with $j \neq i$, and all of its superterms in t are *i-terms*. An *i-term* or *i-literal* is *pure* (or also, *i-pure*) if it only contains symbols from $\Sigma_i \cup C$. Note that the constants in C are the only terms that are both a 1-term and a 2-term, and that an equation is pure

²⁴This means that any literal $(\neg)s = t$ where s and t are not both *i-terms* for some i is considered to be a 1-term. This choice is arbitrary and immaterial.

whenever one of its members is a constant of C and the other is a pure term. The purification procedure consists of the following steps.

Purification Procedure. Let φ be a conjunction of Σ -literals.

1. **Abstract alien subterms.** Apply to completion the following transformation to φ : replace an alien subterm t of a literal of φ with a fresh constant c from C and add (conjunctively) the equation $c = t$ to φ .
2. **Separate.** For $i = 1, 2$, let φ_i be the conjunctions of all the i -literals in (the new) φ .²⁵

It is not hard to see that this procedure always terminates, runs in time linear in the size of φ , and produces a formula $\varphi_1 \wedge \varphi_2$ that is equisatisfiable with φ . More precisely, every model of $\varphi_1 \wedge \varphi_2$ is also a model of φ , and for every model \mathcal{A} of φ there is a model of $\varphi_1 \wedge \varphi_2$ that differs from \mathcal{A} at most in the interpretation of the new constants introduced in the abstraction step above.

The constraint propagation mechanism of the original Nelson-Oppen procedure can be abstracted by a preliminary non-deterministic guess of a truth assignment for all possible *interface equalities*, that is, equations between the (uninterpreted) constants shared by φ_1 and φ_2 . To describe that, it is convenient to introduce the following notion [TH96].

Let R be any equivalence relation over a finite set S of terms. The *arrangement of S induced by R* is the formula

$$ar_R(S) := \bigwedge_{(s,t) \in R} (s = t) \wedge \bigwedge_{(s,t) \notin R} s \neq t$$

containing all equations between R -equivalent terms and all disequations between non- R -equivalent terms of S .

The combination procedure below just guesses an arrangement of the shared constants in a pure form $\varphi_1 \wedge \varphi_2$ of φ , adds it to each φ_i , and then asks the corresponding theory solver to check the satisfiability of the extended φ_i .

The Combination Procedure. Let φ be a conjunction of Σ -literals.

1. **Purify input.** For $i = 1, 2$, let φ_i be the i -pure part of φ 's pure form.
2. **Guess an arrangement.** Where C_0 is the set of all the constant symbols occurring in both φ_1 and φ_2 , choose an arrangement $ar_R(C_0)$.
3. **Check pure formulas.** Return “satisfiable” if $\varphi_i \wedge ar_R(C_0)$ is \mathcal{T}_i -satisfiable for $i = 1$ and $i = 2$; return “unsatisfiable” otherwise.

We call any arrangement guessed in Step 2 of the procedure above an *arrangement for φ* . The procedure is trivially terminating, even if all possible arrangements are considered in step 2.²⁶ It is also refutationally sound for *any*

²⁵Note that (dis)equations between constants of C end up in both φ_1 and φ_2 .

²⁶Observe that there is always a finite, albeit possibly very large, set of arrangements for any finite set of terms.

signature-disjoint theories \mathcal{T}_1 and \mathcal{T}_2 : for such theories, φ is $\mathcal{T}_1 \oplus \mathcal{T}_2$ -unsatisfiable if the procedure returns “unsatisfiable” for every possible arrangement for φ . The procedure, however, is not complete without further assumptions on \mathcal{T}_1 and \mathcal{T}_2 ; that is, in general there could be arrangements for which it returns “satisfiable” even if φ is in fact $\mathcal{T}_1 \oplus \mathcal{T}_2$ -unsatisfiable. A sufficient condition for completeness is that both theories be *stably infinite* [Opp80b].

Definition 33.6.1. Let Σ be a signature and C an infinite set of constants not in Σ . A Σ -theory T is *stably infinite* if every \mathcal{T} -satisfiable ground formula of signature $\Sigma \cup C$ is satisfiable in a model of \mathcal{T} with an infinite universe.

Proposition 33.6.2 (Soundness and completeness). If \mathcal{T}_1 and \mathcal{T}_2 are signature-disjoint and both stably infinite, then the combination procedure returns “unsatisfiable” for an input φ and every possible arrangement for φ iff φ is $\mathcal{T}_1 \oplus \mathcal{T}_2$ -unsatisfiable.

The first (correct) proof of the result above was given in [Opp80b]. More recent proofs based on model-theoretic arguments can be found in [TH96, MZ03b, Ghi04], among others.²⁷ The completeness result in the original paper by Nelson and Oppen [NO79] is incorrect as stated because it imposes no restrictions on the component theories other than that they should be signature-disjoint. The completeness proof however is incorrect only because it implicitly assumes there is no loss of generality in considering only infinite models, which is however not the case unless the theories are stably infinite.

We remark that the stable infiniteness requirement is not an artifice of a particular completeness proof. The combination method is really incomplete without it, as shown for instance in [TZ05]. This is a concern because non-stably infinite theories of practical relevance do exist. For instance, all the theories of a single finite model, such as the theory of bit-vectors (or of strings) with some maximum size, are not stably infinite.

Current practice and recent theoretical work have shown that it is possible to lift the stable infiniteness requirement in a number of ways provided that the combination method is modified to include the propagation of certain cardinality constraints, in addition to equality constraints between shared constants [FG04, TZ05, FRZ05, RRZ05, KGGT07]. More importantly, recent research has shown (see [FRZ05, RRZ05, KGGT07]) that the vexation of the requirement is greatly reduced or disappears in practice if one frames SMT problems within a sorted (i.e., typed) logic, a more natural kind of logic for SMT applications than the classical, unsorted logic traditionally used. The main idea is that, in SMT, combined theories are usually obtained by combining the theory of some parametric data type $T(\alpha_1, \dots, \alpha_n)$ with the theories of particular instances of the type parameters $\alpha_1, \dots, \alpha_n$. A simple example would be the combined theory of lists of real numbers, say, where the type $\text{List}(\text{Real})$ is obtained by instantiating with the type Real the parameter α in the parametric type $\text{List}(\alpha)$.

With combined theories obtained by type parameter instantiation, the theories of the instantiated parameters can be arbitrary, as long as entailed cardinality constraints on the parameter types are propagated properly between theory

²⁷While those papers consider only theories specified by a set of axioms, their proofs also apply to theories specified as sets of models.

solvers. For instance, in the theory of lists of Booleans, say, a pure formula φ_1 of the form

$$l_1 \neq l_2 \wedge l_1 \neq l_3 \wedge l_2 \neq l_3 \wedge \text{tail}(l_1) = \text{nil} \wedge \text{tail}(l_2) = \text{nil} \wedge \text{tail}(l_3) = \text{nil},$$

stating that l_1, l_2, l_3 are distinct lists of length 1, entails the existence of at least 3 distinct values for the element type `Bool`. If φ_1 above is part of the formula sent to the list solver, its entailed minimal cardinality constraint on the `Bool` type must be communicated to the theory solver for that type. Otherwise, the unsatisfiability of φ_1 in the combined theory of `List(Bool)` will go undetected. A description of how it is possible to propagate cardinality constraints conveniently for theories of typical parametric datatypes, such as lists, tuples, arrays and so on, can be found in [RRZ05]. A discussion on why in a typed setting parametricity and not stable infiniteness is the key notion to consider for Nelson-Oppen style combination is provided in [KGGT07].

33.6.2. The Nelson-Oppen Procedure

The original Nelson-Oppen procedure can be seen as a concrete, and improved, implementation of the non-deterministic procedure in the previous subsection.

The first concrete improvement concerns the initial purification step on the input formula. Actually purifying the input formula is unnecessary in practice provided that each theory solver accepts literals containing alien subterms, and treats the latter as if they were free constants. In that case, interface equalities are actually equalities between certain alien subterms.²⁸ A description of the Nelson-Oppen procedure without the purification step is provided in [BDS02b].

A more interesting improvement is possible when each component theory \mathcal{T}_i is convex—as defined in §33.2.1.2. Then, it is enough for each theory solver to propagate recursively and to completion all the interface equalities entailed by its current pure half of the input formula. In concrete, this requires each solver to be able to infer entailed interface equalities and to pass them to the other solver (for addition to its pure half) until one of the two detects an unsatisfiability, or neither has any new equalities to propagate. In the latter case, it is safe to conclude that the input formula is satisfiable in the combined theory.

When one of the two theories, or both, are non-convex, exchanging just entailed interface equalities is no longer enough for completeness. The solver for the non-convex theory must be able to infer, and must propagate, also any *disjunction* of interface equality entailed by its pure half of the formula. Correspondingly, the other solver must be able to process such disjunctions, perhaps by case splits, in addition to the conjunction of literals in the original input.

For convex theories—such as, for instance $\mathcal{T}_{\mathcal{E}}$ —computing the interface equalities entailed by a conjunction φ of literals can be done very efficiently, typically as a by-product of checking φ 's satisfiability (see, e.g., [NO03, NO05b]). For non-convex theories, on the other hand, computing entailed disjunctions of equalities can be rather expensive both in practice and in theory. For instance, for the difference logic fragment of $\mathcal{T}_{\mathcal{Z}}$, this entailment problem is NP-complete even if the satisfiability of conjunctions of literals can be decided in polynomial time [LM05].

²⁸While this simple fact was well understood by earlier implementors of the Nelson-Oppen procedure—including its authors—it was (and still is) often overlooked by casual readers.

33.6.3. Delayed Theory Combination

Delayed Theory Combination (DTC) is a general method for tackling the problem of theory combination within the context of lazy SMT [BBC⁺05c, BBC⁺06]. As in §33.6.1, we assume that $\mathcal{T}_1, \mathcal{T}_2$ are two signature-disjoint stably-infinite theories with their respective \mathcal{T}_i -solvers. Importantly, no assumption is made about the capability of the \mathcal{T}_i -solvers of deducing (disjunctions of) interface equalities from the input set of literals (e_{ij} -deduction capabilities, see §33.4.1): for each \mathcal{T}_i -solver, every intermediate situation from complete e_{ij} -deduction (like in deterministic Nelson-Oppen) to no e_{ij} -deduction capabilities (like in non-deterministic Nelson-Oppen) is admitted.

In a nutshell, in DTC the embedded DPLL engine not only enumerates truth-assignments for the atoms of the input formula, but also “nondeterministically guesses” truth values for the equalities that the \mathcal{T} -solvers are not capable of inferring, and handles the case-split induced by the entailment of disjunctions of interface equalities in non-convex theories. The rationale is to exploit the full power of a modern DPLL engine by delegating to it part of the heavy reasoning effort previously assigned to the \mathcal{T}_i -solvers.

An implementation of DTC [BBC⁺06, BCF⁺06b] is based on the online integration schema of Figure 33.3, exploiting early pruning, \mathcal{T} -propagation, \mathcal{T} -backjumping and \mathcal{T} -learning. Each of the two \mathcal{T}_i -solvers interacts only with the DPLL engine by exchanging literals via the truth assignment μ in a stack-based manner, so that there is no direct exchange of information between the \mathcal{T}_i -solvers. Let \mathcal{T} be $\mathcal{T}_1 \cup \mathcal{T}_2$. The \mathcal{T} -DPLL algorithm is modified in the following ways [BBC⁺06, BCF⁺06b]:²⁹

- \mathcal{T} -DPLL must be instructed to assign truth values not only to the atoms in φ , but also to the interface equalities not occurring in φ . $\mathcal{B}2\mathcal{T}$ and $\mathcal{T}2\mathcal{B}$ are modified accordingly. In particular, \mathcal{T} -`decide_next_branch` is modified to select also new interface equalities not occurring in the original formula.
- μ^p is partitioned into three components $\mu_{\mathcal{T}_1}^p, \mu_{\mathcal{T}_2}^p$ and μ_e^p , s.t. $\mu_{\mathcal{T}_i}$ is the set of i -pure literals and μ_e is the set of interface (dis)equalities in μ .
- \mathcal{T} -`deduce` is modified to work as follows: for each \mathcal{T}_i , $\mu_{\mathcal{T}_i}^p \cup \mu_e^p$ is fed to the respective \mathcal{T}_i -solver. If both return `Sat`, then \mathcal{T} -`deduce` returns `Sat`, otherwise it returns `Conflict`.
- Early-pruning is performed; if some \mathcal{T}_i -solver can deduce atoms or single interface equalities, then \mathcal{T} -propagation is performed. If one \mathcal{T}_i -solver performs the deduction $\mu^* \models_{\mathcal{T}_i} \bigvee_{j=1}^k e_j$, s.t. $\mu^* \subseteq \mu_{\mathcal{T}_i} \cup \mu_e$, each e_j being an interface equality, then the deduction clause $\mathcal{T}2\mathcal{B}(\mu^* \rightarrow \bigvee_{j=1}^k e_j)$ is learned.
- \mathcal{T} -`analyze_conflict` and \mathcal{T} -`backtrack` are modified so as to use the conflict set returned by one \mathcal{T}_i -solver for \mathcal{T} -backjumping and \mathcal{T} -learning. Importantly, such conflict sets may contain interface equalities.

In order to achieve efficiency, other heuristics and strategies have been further suggested in [BBC⁺05c, BBC⁺06, BCF⁺06b, DdM06a, dMB07].

Example 33.6.3. [BCF⁺06b] Consider the set of $\mathcal{T}_{\mathcal{E}} \cup \mathcal{T}_{\mathcal{Z}}$ -literals $\mu =_{def} \mu_{\mathcal{E}} \wedge \mu_{\mathcal{Z}}$ of Figure 33.6. We assume that both the $\mathcal{T}_{\mathcal{E}}$ and $\mathcal{T}_{\mathcal{Z}}$ solvers have no e_{ij} -deduction

²⁹For simplicity, we assume φ is pure, although this condition is not necessary, as in [BDS02b].

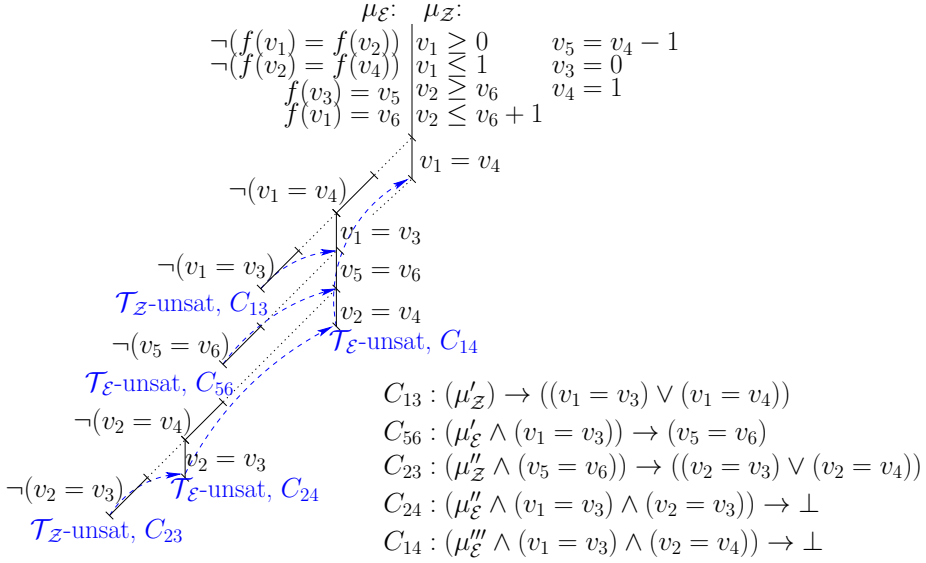


Figure 33.6. The DTC search tree for Example 33.6.3 on $\mathcal{T}_Z \cup \mathcal{T}_E$, with no e_{ij} -deduction. v_1, \dots, v_6 are interface terms. $\mu'_{\mathcal{T}_i}, \mu''_{\mathcal{T}_i}, \mu'''_{\mathcal{T}_i}$ denote appropriate subsets of $\mu_{\mathcal{T}_i}$, $\mathcal{T}_i \in \{\mathcal{T}_E, \mathcal{T}_Z\}$.

capabilities. For simplicity, we also assume that both \mathcal{T}_i -solvers always return conflict sets which do not contain redundant interface disequalities $\neg e_{ij}$. (We adopt here a strategy for DTC which is described in detail in [BCF⁺06b].) In short, \mathcal{T} -DPLL performs a Boolean search on the e_{ij} 's, backjumping on the conflicting clauses $C_{13}, C_{56}, C_{23}, C_{24}$ and C_{14} , which in the end causes the unit-propagation of $(v_1 = v_4)$. Then, \mathcal{T} -DPLL selects a sequence of $\neg e_{ij}$'s without generating conflicts, and concludes that the formula is $\mathcal{T}_1 \cup \mathcal{T}_2$ -satisfiable. Notice that the backjumping steps on the clauses C_{13}, C_{56} , and C_{23} mimic the effects of performing e_{ij} -deductions.

By adopting \mathcal{T} -solvers with different e_{ij} -deduction power, one can trade part or all the e_{ij} -deduction effort for extra Boolean search. [BCF⁺06b] shows that, if the \mathcal{T} -solvers have full e_{ij} -deduction capabilities, then no extra Boolean search on the e_{ij} 's is required; otherwise, the Boolean search is controlled by the quality of the conflict sets returned by the \mathcal{T} -solvers: the more redundant interface disequalities are removed from the conflict sets, the more Boolean branches are pruned. If the conflict sets do not contain redundant interface disequalities, the extra effort is reduced to one branch for each deduction saved, as in Example 33.6.3.

As seen in §33.5.2, the idea from DTC of delegating to the DPLL engine part or most of the, possibly very expensive, reasoning effort normally assigned to the \mathcal{T}_i -solvers (e_{ij} -deduction, case-splits) is pushed even further in the splitting on demand approach. As far as multiple theories are concerned, the latter approach differs from DTC in the fact that the interaction is controlled by the \mathcal{T}_i -solvers, not by the DPLL engine. Other improvements of DTC are currently implemented

in the MATHSAT [BBC⁺06], YICES [DdM06a], and Z3 [dMB07] lazy SMT tools. In particular, [DdM06a] introduced the idea of generating e_{ij} 's on-demand, and [dMB07] that of having the Boolean search on e_{ij} 's driven by a model under construction.

33.6.4. Ackermann's expansion

When combining one or more theories with \mathcal{T}_E , one possible approach is to eliminate uninterpreted function symbols by means of Ackermann's expansion [Ack54]. The method works as described in §33.3.1.3 by replacing every function application occurring in the input formula φ with a fresh variable and then adding to φ all the needed functional congruence constraints. The formula φ' obtained is equisatisfiable with φ , and contains no uninterpreted function symbols. [BCF⁺06a] presents a comparison between DTC and Ackermann's expansion.

Example 33.6.4. Consider the $\mathcal{T}_E \cup \mathcal{T}_Z$ conjunction μ of Example 33.6.3 [BCF⁺06b]. Applying Ackermann's expansion we obtain the conjunction of \mathcal{T}_Z -literals:

$$\begin{aligned} \mu_E &: \neg(v_f(v_1) = v_f(v_2)) \wedge \neg(v_f(v_2) = v_f(v_4)) \wedge (v_f(v_3) = v_5) \wedge (v_f(v_1) = v_6) \wedge \\ \mu_Z &: (v_1 \geq 0) \wedge (v_1 \leq 1) \wedge (v_5 = v_4 - 1) \wedge (v_3 = 0) \wedge (v_4 = 1) \wedge \\ &\quad (v_2 \geq v_6) \wedge (v_2 \leq v_6 + 1) \wedge \\ \text{Ack} &: ((v_1 = v_2) \rightarrow (v_f(v_1) = v_f(v_2))) \wedge ((v_1 = v_3) \rightarrow (v_f(v_1) = v_f(v_3))) \wedge \\ &\quad ((v_1 = v_4) \rightarrow (v_f(v_1) = v_f(v_4))) \wedge ((v_2 = v_3) \rightarrow (v_f(v_2) = v_f(v_3))) \wedge \\ &\quad ((v_2 = v_4) \rightarrow (v_f(v_2) = v_f(v_4))) \wedge ((v_3 = v_4) \rightarrow (v_f(v_3) = v_f(v_4))), \end{aligned} \tag{33.2}$$

which every \mathcal{T}_Z -solver finds \mathcal{T}_Z -satisfiable. (E.g., the \mathcal{T}_Z -model $v_2 = 2$, $v_3 = v_5 = v_6 = v_f(v_1) = v_f(v_3) = 0$, $v_1 = v_4 = v_f(v_2) = v_f(v_4) = 1$ satisfies it.)

33.7. Extensions and Enhancements

33.7.1. Combining eager and lazy approaches

The Ackermann expansion described above is one way to combine eager and lazy approaches. Other hybrids of eager and lazy encoding methods can also be effective.

For instance, consider the satisfiability problem for integer linear arithmetic and the small-domain encoding technique presented in §33.3.2. Due to the conservative nature of the bound derived, and in spite of the many optimizations possible, the computed solution bound can generate a SAT problem beyond the reach of current solvers. For example, this situation can arise for problem domains that do not generate sparse linear constraints.

One can observe that the derived bounds are dependent only on the “bag of constraints”, rather than on their specific Boolean combination in the input formula. Thus, there is hope that a smaller solution bound might suffice. Kroening et al. [KOSS04] have presented an approach to compute the solution bound incrementally, starting with a small bound and increasing it “on demand”. Figure 33.7 outlines this *lazy* approach to computing the solution bound. Given a \mathcal{T}_Z -formula

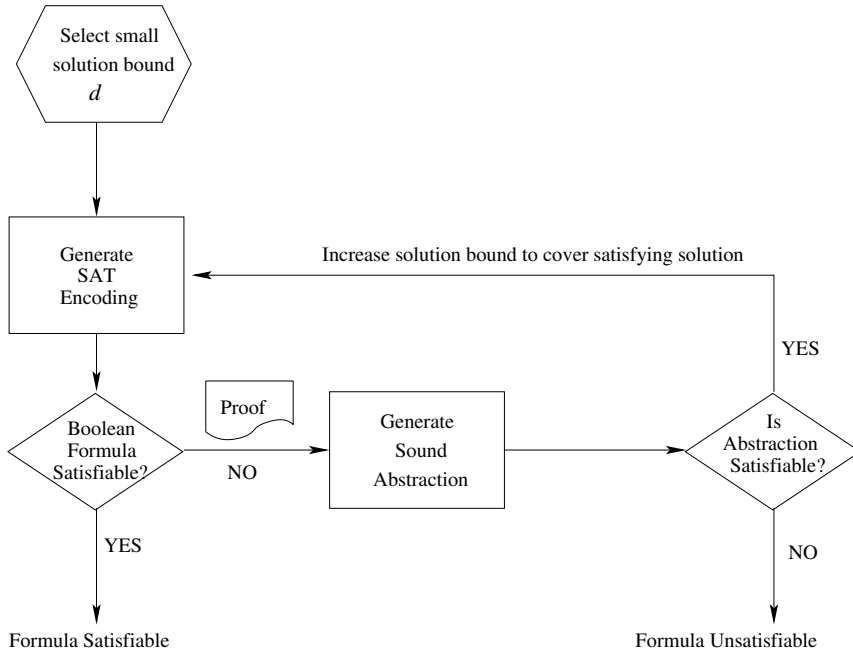


Figure 33.7. Lazy approach to computing solution bound

$F_{\mathcal{Z}}$, we start with an encoding size for each integer variable that is smaller than that prescribed by the conservative bound (say, for example, 1 bit per variable).

If the resulting Boolean formula is satisfiable, so is $F_{\mathcal{Z}}$. If not, the proof of unsatisfiability generated by the SAT solver is used to generate a *sound abstraction* $F'_{\mathcal{Z}}$ of $F_{\mathcal{Z}}$. A sound abstraction is a formula, usually much smaller than the original, such that if it is unsatisfiable, so is the original formula. A sound and complete decision procedure for quantifier-free formulas in $\mathcal{T}_{\mathcal{Z}}$ is then used on $F'_{\mathcal{Z}}$. If this decision procedure concludes that $F'_{\mathcal{Z}}$ is unsatisfiable, so is $F_{\mathcal{Z}}$. If not, it provides a counterexample which indicates the necessary increase in the encoding size. A new SAT-encoding is generated, and the procedure repeats.

The bound S on solution size that is derived in §33.3.2 implies an upper bound nS on the number of iterations of this lazy encoding procedure; thus the lazy encoding procedure needs only polynomially many iterations before it terminates with the correct answer. Of course, each iteration involves a call to a SAT solver as well as to the $\mathcal{T}_{\mathcal{Z}}$ -solver.

A key component of this lazy approach is the generation of the sound abstraction. While the details are outside the scope of this chapter, we sketch one approach here. (Details can be found in [KOSS04].) Assume that $F_{\mathcal{Z}}$ is in conjunctive normal form (CNF); thus, $F_{\mathcal{Z}}$ can be viewed as a set of clauses, each of which is a disjunction of linear constraints and Boolean literals. A subset of this set of clauses is a sound abstraction of $F_{\mathcal{Z}}$. This subset is computed by retaining only those clauses from the original set that contribute to the proof of unsatisfiability of the SAT-encoding.

If the generated abstractions are small, the sound and complete decision procedure used by this approach will run much faster than if it were fed the original formula. Thus, one can view this approach as an “accelerator” that can speed up any SMT solver. The approach is also applicable to other theories; for instance, it has been successfully extended to finite-precision bit-vector arithmetic [BKO⁺07].

33.7.2. Handling quantifiers

Because the Nelson-Oppen framework (§33.6.1) only works for quantifier-free formulas, SMT solvers have traditionally been rather limited in their ability to reason about quantifiers. A notable exception is the prover Simplify [DNS03] which uses *quantifier instantiation* on top of a Nelson-Oppen style prover. Several modern SMT solvers have adopted and extended these techniques [GBT07, BdM07, ML06].

The basic idea can be understood by extending the abstract framework described in §33.4.4 to include rules for quantifier instantiation. The main modification is to also allow closed quantified formulas wherever atomic formulas are allowed. Define a *generalized atomic formula* as either an atomic formula or a closed quantified formula. A *generalized literal* is either a generalized atomic formula or its negation; a *generalized clause* is a disjunction of generalized literals. The modified abstract framework is obtained simply by allowing literals and clauses to be replaced by their generalized counterparts. For instance, non-fail states become pairs $M \parallel F$, where M is a sequence of generalized literals and F is a conjunction of generalized clauses. With this understanding, the two rules below can be used to model quantifier instantiation. For simplicity and without loss of generality, we assume here that quantified literals in M appear only positively and that the bodies of quantified formulas are themselves in abstract CNF.

$$\begin{aligned} \exists\text{-Inst} : \quad M \parallel F &\implies M \parallel F, \neg\exists\mathbf{x}.\varphi \vee \varphi[\mathbf{x}/\mathbf{a}] \quad \text{if } \begin{cases} \exists\mathbf{x}.\varphi \text{ is in } M \\ \mathbf{a} \text{ are fresh constants} \end{cases} \\ \forall\text{-Inst} : \quad M \parallel F &\implies M \parallel F, \neg\forall\mathbf{x}.\varphi \vee \varphi[\mathbf{x}/\mathbf{s}] \quad \text{if } \begin{cases} \forall\mathbf{x}.\varphi \text{ is in } M \\ \mathbf{s} \text{ are ground terms} \end{cases} \end{aligned}$$

The $\exists\text{-Inst}$ rule essentially models skolemization and the $\forall\text{-Inst}$ rule models instantiation. It is also clear that termination can only be guaranteed by limiting the number of times the rules are applied. For a given existentially quantified formula, there is no benefit to applying $\exists\text{-Inst}$ more than once, but a universally quantified formula may need to be instantiated several times with different ground terms before unsatisfiability is detected. The main challenge then, in applying these rules, is to come up with an effective strategy for applying $\forall\text{-Inst}$. For some background theories (e.g., universal theories) completeness can be shown for exhaustive and fair instantiation strategies. In practice, however, the most effective techniques are incomplete and heuristic.

The most common approach is to select for instantiation only ground terms that are *relevant* to the quantified formula in question, according to some heuristic

relevance criteria. The idea is as follows: given a state $M \parallel F$ and an abstract literal $\forall x.\varphi$ in M , try to find a subterm t of $\forall x.\varphi$ properly containing x , a ground term g in M , and a subterm s of g , such that $t[x/s]$ is equivalent to g modulo the background theory \mathcal{T} (written $t[x/s] =_{\mathcal{T}} g$). In this case, we expect that instantiating x with s is more likely to be helpful than instantiating with other candidate terms. Following Simplify’s terminology, the term t is called a *trigger* (for $\forall x.\varphi$). In terms of unification theory, the case in which $t[x/s] =_{\mathcal{T}} g$ is a special case of \mathcal{T} -matching between t and g .

In general, in the context of SMT, given the complexity of the background theory \mathcal{T} , it may be very difficult if not impossible to determine whether a trigger and a ground term \mathcal{T} -match. As a result, most implementations use some form of $\mathcal{T}_{\mathcal{E}}$ -matching. For details on effective implementation strategies, we refer the reader to [DNS03, GBT07, BdM07, ML06].

33.7.3. Producing models

For some applications, it is desirable not only to know whether a formula is satisfiable, but if so, what a satisfying model is. In general, it may be challenging to capture all of the structure of an arbitrary first-order model. However, it is often sufficient to know how to assign values from some “standard” model to the ground terms in a formula. Several SMT solvers have implemented support for “models” of this kind.

One approach is to do additional work guessing values for ground terms and then to double-check that the formula is indeed satisfied. This is the approach followed by CVC3 [BT07].

An alternative approach is to instrument the \mathcal{T} -solver to continually maintain a value for every ground term associated with the theory. This is the strategy followed by the solvers YICES and Z3 [DdM06b, dMB07].

33.7.4. Producing proofs

In both SAT and SMT communities, the importance of having tools able to produce proofs of the (\mathcal{T} -)unsatisfiability of the input formulas has been recently stressed, due to the need for independently verifiable results, and as a starting point for the production of unsatisfiable cores (§33.7.5) and interpolants (§33.7.6).

A DPLL solver can be easily modified to generate a resolution proof of unsatisfiability for an input CNF formula, by keeping track of all resolution steps performed when generating each conflict clause, and by combining these subproofs iteratively into a proof of unsatisfiability whose leaves are original clauses. Techniques for translating a Boolean resolution proof into a more human-readable and verifiable format have been proposed, e.g., in [BB03, Amj07].

Similarly, a lazy SMT solver can be modified to generate a resolution proof of unsatisfiability for an input CNF formula, whose leaves are either clauses from the original formula or \mathcal{T} -lemmas (i.e. negations of \mathcal{T} -conflict sets and \mathcal{T} -deduction clauses returned by the \mathcal{T} -solver, see §33.4.3), which are \mathcal{T} -valid clauses. Such a resolution proof can be further refined by providing a proof for each \mathcal{T} -lemma in some theory-specific deductive framework (like, e.g. that in [McM05] for inequalities in $\mathcal{T}_{\mathcal{R}}$).

33.7.5. Identifying unsatisfiable cores

An *unsatisfiable core* (*UC*) of an unsatisfiable CNF formula φ is an unsatisfiable subset of the clauses in φ .

In SAT, the problem of finding small unsatisfiable cores has been addressed by many authors in recent years [LMS04, MLA⁺05, ZM03, OMA⁺04, Hua05, DHN06, Boo, GKS06, ZLS06] due to its importance in formal verification [McM02]. In particular, lots of techniques and optimizations have been introduced with the aim of producing small [ZM03, GKS06], minimal [OMA⁺04, Hua05, DHN06], or even minimum unsat cores [LMS04, MLA⁺05, ZLS06].

In SMT, despite the fact that finding unsatisfiable cores has been addressed explicitly in the literature only recently [CGS07], at least three SMT solvers (i.e. CVC3, MATHSAT, and YICES) support UC generation.³⁰ We distinguish three main approaches.

In the approach implemented in CVC3 and MATHSAT (*proof-based approach* hereafter), the UC is produced as a byproduct of the generation of resolution proofs. As in [ZM03], the idea is to return the set of clauses from the original problem which appear as leaves in the resolution proof of unsatisfiability. (\mathcal{T} -lemmas are not included as they are \mathcal{T} -valid clauses, so they play no role in the \mathcal{T} -unsatisfiability of the core.)

The approach used by YICES (*assumption-based approach* hereafter) is an adaptation of a method used in SAT [LMS04]: for each clause C_i in the problem, a new Boolean “selector” variable S_i is created; then, each C_i is replaced by $(S_i \rightarrow C_i)$; finally, before starting the search each S_i is forced to true. In this way, when a conflict at decision level zero is found by the DPLL solver, the conflict clause C contains only selector variables, and the UC returned is the union of the clauses whose selectors appear in C . Neither approach aims at producing minimal or minimum unsatisfiable cores, nor does anything to reduce their size.

In the *lemma-lifting approach* [CGS07] implemented in MATHSAT, a lazy SMT solver is combined with an external (and possibly highly-optimized) propositional core extractor. The SMT solver stores and returns the \mathcal{T} -lemmas it had to prove in order to refute the input formula; the external core extractor is then called on the Boolean abstraction of the original SMT problem and of the \mathcal{T} -lemmas. Clauses corresponding to \mathcal{T} -lemmas are removed from the resulting UC, and the remaining abstract clauses are refined back into their original form. The result is an unsatisfiable core of the original SMT problem. This technique is able to benefit from any size-reduction techniques implemented in the Boolean core extractor used.

33.7.6. Computing interpolants

A *Craig interpolant* (“interpolant” hereafter) of a pair of formulas (ψ, ϕ) s.t. $\psi \wedge \phi \models_{\mathcal{T}} \perp$ is a formula ψ' s.t.:

- $\psi \models_{\mathcal{T}} \psi'$,

³⁰Apart from [CGS07], the information reported here on the computation of unsatisfiable cores in these tools comes from personal knowledge of the tools (CVC3 and MATHSAT) and from private communications from the authors (YICES).

- $\psi' \wedge \phi \models_{\mathcal{T}} \perp$, and
- $\psi' \preceq \psi$ and $\psi' \preceq \phi$,

where $\alpha \preceq \beta$ denotes that all uninterpreted (in the signature of \mathcal{T}) symbols of α appear in β . Note that a quantifier-free interpolant exists if \mathcal{T} admits quantifier elimination [KMZ06] (e.g., in $\mathcal{T}_{\mathcal{Z}}$ a quantifier-free interpolant for (ψ, ϕ) may not exist).

The use of interpolation in formal verification was introduced by McMillan in [McM03] for purely-propositional formulas, and it was subsequently extended to handle $\mathcal{T}_{\mathcal{E}} \cup \mathcal{T}_{\mathcal{R}}$ -formulas in [McM05]. The technique, which is based on earlier work by Pudlák [Pud97], works as follows. (We assume w.l.o.g. that the formulas are in CNF.) An interpolant for (ψ, ϕ) is constructed from a resolution proof \mathcal{P} of the unsatisfiability of $\psi \wedge \phi$, according to the following steps:

1. for every clause C occurring as a leaf in \mathcal{P} , set $I_C \equiv C \downarrow \phi$ if $C \in \psi$, and $I_C \equiv \top$ if $C \in \phi$;
2. for every \mathcal{T} -lemma $\neg\eta$ occurring as a leaf in \mathcal{P} , generate an interpolant $I_{\neg\eta}$ for $(\eta \setminus \phi, \eta \downarrow \phi)$;
3. for every node C of \mathcal{P} obtained by resolving $C_1 \equiv p \vee \phi_1$ and $C_2 \equiv \neg p \vee \phi_2$, set $I_C \equiv I_{C_1} \vee I_{C_2}$ if p does not occur in ϕ , and $I_C \equiv I_{C_1} \wedge I_{C_2}$ otherwise;
4. return I_{\perp} as an interpolant for (ψ, ϕ) ;

where $C \downarrow \phi$ is the clause obtained by removing all the literals in C whose atoms do not occur in ϕ , and $C \setminus \phi$ that obtained by removing all the literals whose atoms do occur in ϕ . In the purely-Boolean case the algorithm reduces to steps 1., 3. and 4. only. Notice that step 2 of the algorithm is the only part which depends on the theory \mathcal{T} , so that the problem reduces to that of finding interpolants for (negated) \mathcal{T} -lemmas.

A number of techniques exist for theory-specific interpolation generation. For example, [McM05] provides a set of rules for constructing interpolants for \mathcal{T} -lemmas in $\mathcal{T}_{\mathcal{E}}$, for weak linear inequalities ($0 \leq t$) in $\mathcal{T}_{\mathcal{R}}$, and their combination. [YM05] uses a variant of the Nelson-Oppen procedure (§33.6.2) for generating interpolants for $\mathcal{T}_1 \cup \mathcal{T}_2$ using the interpolant-generation procedures of \mathcal{T}_1 and \mathcal{T}_2 as black-boxes. The combination $\mathcal{T}_{\mathcal{E}} \cup \mathcal{T}_{\mathcal{R}}$ can also be used to compute interpolants for other theories, such as those of lists, arrays, sets and multisets [KMZ06]. [RSS07] computes interpolants for \mathcal{T} -lemmas in $\mathcal{T}_{\mathcal{E}} \cup \mathcal{T}_{\mathcal{R}}$ by solving a system of Linear Programming (LP) constraints. [KW07] extends the *eager* SMT approach to the generation of interpolants. (The approach is currently limited to the theory of equality only, without uninterpreted functions.) [CGS08] presents some extensions to the work in [McM05], including an optimized interpolant generator for the full theory $\mathcal{T}_{\mathcal{R}}$, an ad hoc interpolant generator for difference logic, and an interpolant combination method based on Delayed Theory Combination (§33.6.3).

33.7.7. Recent advances

In the decade since the first version of this book chapter was written, the field of SMT solving has matured considerably. The literature in the intervening period

is too large for us to fully review here. We focus on just three directions that are producing exciting advances.

New theories: SMT solving has been extended to new background theories such as that of strings with operations such as concatenation, length, and membership in regular languages [LRT⁺16, BGZ17]. Even for existing theories, researchers are finding special cases where domain-specific structure can be exploited to obtain several orders of magnitude speed-up in solver time: a good example is solving Boolean combinations of (possibly non-linear) convex constraints over the reals [NPSSV10, SNSV⁺17, SNS⁺18].

From satisfiability to optimization: A natural extension of SMT solving is in the direction of optimization. In these variants, termed *optimization modulo theories*, one seeks to optimize a cost function while satisfying constraints represented as an SMT formula. While initial promising results have been obtained for special cases [MPP15, ST17], much more remains to be done on this topic.

SMT and synthesis: One of the exciting new applications of SMT solving in the past decade has been to the algorithmic synthesis of programs and other formal artifacts [JGST10, Ses15]. Recently several synthesis problems have been unified by the formalism of *syntax-guided synthesis* (SyGuS) [ABJ⁺13]. In the simplest version of SyGuS, one is given an SMT formula φ with an uninterpreted function f to be synthesized and a grammar of expressions for the implementation of this function, and one must determine whether there exists an expression which, when substituted for f in φ , causes it to become valid. The connection between SMT and SyGuS is synergistic. Clearly, determining whether a candidate solution to a SyGuS problem is correct involves solving an SMT problem. Additionally, SyGuS solvers are proving useful in solving synthesis sub-tasks within SMT solving such as generating terms with which to instantiate quantifiers [PNB17, NPR⁺18]. We anticipate this connection between SMT solving and synthesis will only deepen in the years to come.

References

- [ABC⁺02] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions. In *Proc. CADE'2002.*, volume 2392 of *LNAI*. Springer, July 2002.
- [ABJ⁺13] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 1–17, October 2013.
- [ABRS05] A. Armando, M. P. Bonacina, S. Ranise, and S. Schulz. On a rewriting approach to satisfiability procedures: Extension, combination of theories and an experimental appraisal. In B. Gramlich, editor, *Frontiers of Combining Systems, 5th International Workshop, FroCoS 2005, Vienna, Austria, September 19-21, 2005, Proceedings*, volume 3717 of *Lecture Notes in Computer Science*, pages 65–80. Springer, 2005.

- [ACG99] A. Armando, C. Castellini, and E. Giunchiglia. SAT-based procedures for temporal reasoning. In *Proc. European Conference on Planning, ECP-99*, 1999.
- [ACGM04] A. Armando, C. Castellini, E. Giunchiglia, and M. Maratea. A SAT-based Decision Procedure for the Boolean Combination of Difference Constraints. In *Proc. SAT'04*, 2004.
- [Ack54] W. Ackermann. *Solvable Cases of the Decision Problem*. North-Holland, Amsterdam, 1954.
- [ACKS02] G. Audemard, A. Cimatti, A. Kornilowicz, and R. Sebastiani. SAT-Based Bounded Model Checking for Timed Systems. In *Proc. FORTE'02.*, volume 2529 of *LNCS*. Springer, November 2002.
- [AG93] A. Armando and E. Giunchiglia. Embedding Complex Decision Procedures inside an Interactive Theorem Prover. *Annals of Mathematics and Artificial Intelligence*, 8(3-4):475-502, 1993.
- [Amj07] H. Amjad. A Compressing Translation from Propositional Resolution to Natural Deduction. In *Proc. FroCoS*, volume 4720 of *Lecture Notes in Computer Science*, pages 88-102. Springer, 2007.
- [ARR03] A. Armando, S. Ranise, and M. Rusinowitch. A Rewriting Approach to Satisfiability Procedures. *Information and Computation*, 183(2):140-164, June 2003.
- [BB03] C. Barrett and S. Berezin. A proof-producing boolean search engine. In *Proceedings of the 1st International Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR '03)*, July 2003.
- [BB04] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In R. Alur and D. A. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification (CAV '04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515-518. Springer, July 2004.
- [BB09] R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 5505 of *Lecture Notes in Computer Science*, pages 174-177. Springer, 2009.
- [BBC+05a] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. Rossum, S. Schulz, and R. Sebastiani. An incremental and Layered Procedure for the Satisfiability of Linear Arithmetic Logic. In *Proc. TACAS'05*, volume 3440 of *LNCS*. Springer, 2005.
- [BBC+05b] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. Rossum, S. Schulz, and R. Sebastiani. MathSAT: A Tight Integration of SAT and Mathematical Decision Procedure. *Journal of Automated Reasoning*, 35(1-3), October 2005.
- [BBC+05c] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Ranise, and R. Sebastiani. Efficient Satisfiability Modulo Theories via Delayed Theory Combination. In *Proc. CAV 2005*, volume 3576 of *LNCS*. Springer, 2005.
- [BBC+06] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van

- Rossum, S. Ranise, and R. Sebastiani. Efficient Theory Combination via Boolean Search. *Information and Computation*, 204(10), 2006.
- [BCF⁺06a] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, A. Santuari, and R. Sebastiani. To Ackermann-ize or not to Ackermann-ize? On Efficiently Handling Uninterpreted Function Symbols in *SMT($\mathcal{EUF} \cup \mathcal{T}$)*. In *Proc. LPAR'06*, volume 4246 of *LNAI*. Springer, 2006.
- [BCF⁺06b] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. Delayed Theory Combination vs. Nelson-Oppen for Satisfiability Modulo Theories: a Comparative Analysis. In *Proc. LPAR'06*, volume 4246 of *LNAI*. Springer, 2006.
- [BCF⁺07] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, Z. Hanna, A. Nadel, A. Palti, and R. Sebastiani. A lazy and layered SMT(BV) solver for hard industrial verification problems. In W. Damm and H. Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 547–560. Springer-Verlag, July 2007.
- [BCLZ04] T. Ball, B. Cook, S. Lahiri, and L. Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In *Proc. Computer-Aided Verification (CAV)*, volume 3114 of *Lecture Notes in Computer Science*, pages 457–461, 2004.
- [BDL98] C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for bit-vector arithmetic. In *Proceedings of the 35th Design Automation Conference (DAC '98)*, pages 522–527. Association for Computing Machinery, June 1998.
- [BdM07] N. Bjørner and L. de Moura. Efficient E-matching for SMT solvers. In F. Pfenning, editor, *Proceedings of the 21st International Conference on Automated Deduction (CADE '07)*, volume 4603 of *Lecture Notes in Artificial Intelligence*, pages 183–198. Springer-Verlag, July 2007.
- [BDS02a] C. W. Barrett, D. L. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In E. Brinksma and K. G. Larsen, editors, *Proceedings of the 14th International Conference on Computer Aided Verification (CAV '02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 236–249. Springer-Verlag, July 2002.
- [BDS02b] C. W. Barrett, D. L. Dill, and A. Stump. A generalization of Shostak's method for combining decision procedures. In A. Armando, editor, *Proceedings of the 4th International Workshop on Frontiers of Combining Systems (FroCoS '02)*, volume 2309 of *Lecture Notes in Artificial Intelligence*, pages 132–146. Springer, April 2002.
- [BDS02c] C. W. Barrett, D. L. Dill, and A. Stump. A generalization of Shostak's method for combining decision procedures. In A. Armando, editor, *Proceedings of the 4th International Workshop on Frontiers of Combining Systems, FroCoS'2002 (Santa Margherita Ligure, Italy)*, volume 2309 of *Lecture Notes in Computer Science*, pages 132–147, apr 2002.

- [BE07a] M. P. Bonacina and M. Echenim. Rewrite-based decision procedures. In M. Archer, T. B. de la Tour, and C. Munoz, editors, *Proceedings of the Sixth Workshop on Strategies in Automated Deduction (STRATEGIES)*, volume 174(11) of *Electronic Notes in Theoretical Computer Science*, pages 27–45. Elsevier, July 2007.
- [BE07b] M. P. Bonacina and M. Echenim. T-decision by decomposition. In F. Pfenning, editor, *Proceedings of the Twenty-first International Conference on Automated Deduction (CADE)*, volume 4603 of *Lecture Notes in Artificial Intelligence*, pages 199–214. Springer, July 2007.
- [BFT86] I. Borosh, M. Flahive, and L. B. Treybig. Small solutions of linear Diophantine equations. *Discrete Mathematics*, 58:215–220, 1986.
- [BGN⁺06a] M. P. Bonacina, S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Decidability and undecidability results for Nelson-Oppen and rewrite-based decision procedures. In U. Furbach and N. Shankar, editors, *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*, pages 513–527. Springer, 2006.
- [BGN⁺06b] M. P. Bonacina, S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Decidability and undecidability results for Nelson-Oppen and rewrite-based decision procedures. In U. Furbach and N. Shankar, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Computer Science*, pages 513–527, Seattle (WA, USA), 2006. Springer.
- [BGV99] R. E. Bryant, S. German, and M. N. Velev. Exploiting Positive Equality in a Logic of Equality with Uninterpreted Functions. In *Proc. CAV'99*, volume 1633 of *LNCS*. Springer, 1999.
- [BGV01] R. E. Bryant, S. German, and M. N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions on Computational Logic*, 2(1):1–41, January 2001.
- [BGZ17] M. Berzish, V. Ganesh, and Y. Zheng. Z3str3: A string solver with theory-aware heuristics. In *Proceedings of the International Conference on Formal Methods in Computer Aided Design (FMCAD)*, pages 55–59, 2017.
- [BKO⁺07] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. Brady. Deciding bit-vector arithmetic with abstraction. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4424 of *Lecture Notes in Computer Science*, pages 358–372. Springer, 2007.
- [BM88] R. S. Boyer and J. Moore. Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic. *Machine Intelligence*, 11:83–124, 1988.
- [BM90] R. S. Boyer and J. S. Moore. A theorem prover for a computational logic. In M. E. Stickel, editor, *10th International Conference on Au-*

- tomated Deduction (CADE)*, LNAI 449, pages 1–15, Kaiserslautern, FRG, July 24–27, 1990. Springer-Verlag.
- [BNOT06a] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on demand in SAT modulo theories. In M. Hermann and A. Voronkov, editors, *Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR '06)*, volume 4246 of *Lecture Notes in Computer Science*, pages 512–526. Springer, November 2006.
- [BNOT06b] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on demand in SAT Modulo Theories. Technical Report 06-05, Department of Computer Science, University of Iowa, August 2006.
- [Boo] Booleforce, <http://fmv.jku.at/booleforce/>.
- [BP98] N. Bjørner and M. C. Pichora. Deciding fixed and non-fixed size bit-vectors. In *TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 376–392. Springer-Verlag, 1998.
- [BRTS08] C. Barrett, S. Ranise, C. Tinelli, and A. Stump. The SMT-LIB web site, 2008.
- [BST07] C. Barrett, I. Shikhanian, and C. Tinelli. An abstract decision procedure for a theory of inductive data types. *Journal on Satisfiability, Boolean Modeling and Computation*, 3:21–46, 2007.
- [BT76] I. Borosh and L. B. Treybig. Bounds on positive integral solutions of linear Diophantine equations. *Proceedings of the American Mathematical Society*, 55(2):299–304, March 1976.
- [BT07] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer, July 2007.
- [BTV03] L. Bachmair, A. Tiwari, and L. Vigneron. Abstract congruence closure. *Journal of Automated Reasoning*, 31(2):129–168, 2003.
- [BV00] R. E. Bryant and M. N. Velev. Boolean satisfiability with transitivity constraints. In A. Emerson and P. Sistla, editors, *Computer-Aided Verification (CAV 2000)*, LNCS 1855. Springer-Verlag, July 2000.
- [CGS07] A. Cimatti, A. Griggio, and R. Sebastiani. A Simple and Flexible Way of Computing Small Unsatisfiable Cores in SAT Modulo Theories. In *Proc. SAT'07*, volume 4501 of *LNCS*. Springer, 2007.
- [CGS08] A. Cimatti, A. Griggio, and R. Sebastiani. Efficient Interpolant Generation in Satisfiability Modulo Theories. In *Proc. TACAS'08*, volume 4963 of *LNCS*. Springer, 2008.
- [Cha93] V. Chandru. Variable elimination in linear constraints. *The Computer Journal*, 36(5):463–472, August 1993.
- [CK06] S. Conchon and S. Krstić. Strategies for combining decision procedures. *Theoretical Computer Science*, 354, 2006.
- [CLS96] D. Cyrluk, P. Lincoln, and N. Shankar. On Shostak's decision procedure for combinations of theories. In M. McRobbie and J. Slaney, editors, *13th International Conference on Computer Aided Deduction*, volume 1104 of *Lecture Notes in Computer Science*, pages 463–477.

- Springer, 1996.
- [CM06a] S. Cotton and O. Maler. Fast and Flexible Difference Logic Propagation for DPLL(T). In *Proc. SAT'06*, volume 4121 of *LNCS*. Springer, 2006.
- [CM06b] S. Cotton and O. Maler. Satisfiability modulo theory chains with DPLL(T). Unpublished. Available from <http://www.verimag.imag.fr/~maler/>, 2006.
- [CMR97] D. Cyrluk, M. O. Möller, and H. Ruess. An efficient decision procedure for the theory of fixed-size bit-vectors. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97)*, pages 60–71. Springer, 1997.
- [Cra57] W. Craig. Linear reasoning: A new form of the Herbrand-Gentzen theorem. *Journal of Symbolic Logic*, 22(3):250–268, 1957.
- [DdM06a] B. Dutertre and L. de Moura. System Description: Yices 1.0. Proc. on 2nd SMT competition, SMT-COMP'06. Available at yices.cs1.sri.com/yices-smtcomp06.pdf, 2006.
- [DdM06b] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In T. Ball and R. B. Jones, editors, *Proceedings of the 18th International Conference on Computer Aided Verification (CAV '06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer, August 2006.
- [DH88] J. H. Davenport and J. Heintz. Real quantifier elimination is doubly exponential. *Journal of Symbolic Computation*, 5:29–35, 1988.
- [DHN06] N. Dershowitz, Z. Hanna, and A. Nadel. A Scalable Algorithm for Minimal Unsatisfiable Core Extraction. In *Proc. SAT'06*, volume 4121 of *LNCS*. Springer, 2006.
- [dMB07] L. de Moura and N. Bjørner. Model-based Theory Combination. In *Proc. 5th workshop on Satisfiability Modulo Theories, SMT'07*, 2007.
- [dMB08] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*, 2008.
- [dMRS02] L. de Moura, H. Rueß, and M. Sorea. Lemmas on Demand for Satisfiability Solvers. Proc. SAT'02, 2002.
- [DNS03] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Laboratories Palo Alto, 2003.
- [DST80] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *Journal of the Association for Computing Machinery*, 27(4):758–771, October 1980.
- [EFT94] H. D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical logic*. Undergraduate Texts in Mathematics. Springer, New York, second edition edition, 1994.
- [EKM98] J. Elgaard, N. Klarlund, and A. Möller. Mona 1.x: New techniques for WS1S and WS2S. In *Proceedings of the 10th International Conference on Computer Aided Verification (CAV '98)*, volume 1427 of *Lecture Notes in Computer Science*. Springer, 1998.
- [End00] H. B. Enderton. *A Mathematical Introduction to Logic*. Undergrad-

- uate Texts in Mathematics. Academic Press, second edition edition, 2000.
- [FG04] P. Fontaine and E. P. Gribomont. Combining non-stably infinite, non-first order theories. In C. Tinelli and S. Ranise, editors, *Proceedings of the IJCAR Workshop on Pragmatics of Decision Procedures in Automated Deduction, PDPAR*, July 2004.
- [FJOS03] C. Flanagan, R. Joshi, X. Ou, and J. B. Saxe. Theorem Proving Using Lazy Proof Explication. In *Proc. CAV 2003*, LNCS. Springer, 2003.
- [FORS01] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: integrated canonizer and solver. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, 2001.
- [FRZ05] P. Fontaine, S. Ranise, and C. G. Zarba. Combining lists with non-stably infinite theories. In F. Baader and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 3452 of *Lecture Notes in Computer Science*, pages 51–66. Springer-Verlag, 2005.
- [Gan02] H. Ganzinger. Shostak light. In A. Voronkov, editor, *Proceedings of the 18th International Conference on Computer-Aided Deduction (CADE '02)*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 332–346. Springer, July 2002.
- [GBT07] Y. Ge, C. Barrett, and C. Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In F. Pfenning, editor, *Proceedings of the 21st International Conference on Automated Deduction (CADE '07)*, volume 4603 of *Lecture Notes in Artificial Intelligence*, pages 167–182. Springer-Verlag, July 2007.
- [GD07] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In W. Damm and H. Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer-Verlag, July 2007.
- [GGT01] E. Giunchiglia, F. Giunchiglia, and A. Tacchella. SAT Based Decision Procedures for Classical Modal Logics. *Journal of Automated Reasoning*. Special Issue: Satisfiability at the start of the year 2000, 2001.
- [Ghi04] S. Ghilardi. Model theoretic methods in combined constraint satisfiability. *Journal of Automated Reasoning*, 3(3–4):221–249, 2004.
- [GHN⁺04] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast Decision Procedures. In *Proc. CAV'04*, volume 3114 of *LNCS*. Springer, 2004.
- [GKS06] R. Gershman, M. Koifman, and O. Strichman. Deriving Small Unsatisfiable Cores with Dominators. In *Proc. CAV'06*, volume 4144 of *LNCS*. Springer, 2006.
- [GNRZ06] S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Deciding extensions of the theory of arrays by integrating decision procedures and instantiation strategies. In M. Fisher, W. van der Hoek, B. Konev, and A. Lisitsa, editors, *Logics in Artificial Intelligence, 10th Eu-*

- ropean Conference, *JELIA 2006, Liverpool, UK, September 13-15, 2006, Proceedings*, volume 4160 of *Lecture Notes in Computer Science*, pages 177–189. Springer, 2006.
- [GNZ05] S. Ghilardi, E. Nicolini, and D. Zucchelli. A comprehensive framework for combined decision procedures. In B. Gramlich, editor, *Frontiers of Combining Systems, 5th International Workshop, FroCoS 2005, Vienna, Austria, September 19-21, 2005, Proceedings*, volume 3717 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 2005.
- [GS96] F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures - the case study of modal K. In *Proc. CADE'13*, number 1104 in *LNAI*. Springer, 1996.
- [HBJ+14] L. Hadarean, K. Bansal, D. Jovanovic, C. Barrett, and C. Tinelli. A tale of two solvers: Eager and lazy approaches to bit-vectors. In *26th International Conference on Computer-Aided Verification (CAV)*, pages 680–695, 2014.
- [HMNT93] D. Hochbaum, N. Megiddo, J. Naor, and A. Tamir. Tight bounds and 2-approximation algorithms for integer programs with two variables per inequality. *Mathematical Programming*, 62:63–92, 1993.
- [Hor98] I. Horrocks. The FaCT system. In H. de Swart, editor, *Proc. TABLEAUX-98*, volume 1397 of *LNAI*, pages 307–312. Springer, 1998.
- [Hua05] J. Huang. MUP: a minimal unsatisfiability prover. In *Proc. ASP-DAC '05*. ACM Press, 2005.
- [JGST10] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 215–224, May 2010.
- [JLS09] S. Jha, R. Limaye, and S. A. Seshia. Beaver: engineering an efficient SMT solver for bit-vector arithmetic. In *Proc. 21st International Conference on Computer-Aided verification (CAV)*, volume 5643 of *Lecture Notes in Computer Science*, pages 668–674, June 2009.
- [JMSY94] J. Jaffar, M. J. Maher, P. J. Stuckey, and R. H. C. Yap. Beyond finite domains. In *2nd International Workshop on Principles and Practice of Constraint Programming (PPCP'94)*, volume 874 of *Lecture Notes in Computer Science*, pages 86–94, 1994.
- [Kar84] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.
- [KC03] S. Krstić and S. Conchon. Canonization for disjoint unions of theories. In F. Baader, editor, *Proceedings of the 19th International Conference on Computer-Aided Deduction (CADE '03)*, volume 2741 of *Lecture Notes in Artificial Intelligence*, pages 197–211. Springer, August 2003. Miami Beach, FL.
- [KGGT07] S. Krstić, A. Goel, J. Grundy, and C. Tinelli. Combined satisfiability modulo parametric theories. In O. Grumberg and M. Huth, editors, *Proceedings of the 13th International Conference on Tools and Algo-*

- arithms for the Construction and Analysis of Systems (Braga, Portugal)*, volume 4424 of *Lecture Notes in Computer Science*, pages 618–631. Springer, 2007.
- [KM78] R. Kannan and C. L. Monma. On the computational complexity of integer programming problems. In *Optimisation and Operations Research*, volume 157 of *Lecture Notes in Economics and Mathematical Systems*, pages 161–172. Springer-Verlag, 1978.
- [KMZ06] D. Kapur, R. Majumdar, and C. G. Zarba. Interpolation for data structures. In *Proc. SIGSOFT FSE*. ACM, 2006.
- [KOSS04] D. Kroening, J. Ouaknine, S. A. Seshia, and O. Strichman. Abstraction-based satisfiability solving of Presburger arithmetic. In *Proc. 16th International Conference on Computer-Aided Verification (CAV)*, pages 308–320, July 2004.
- [KW07] D. Kroening and G. Weissenbacher. Lifting propositional interpolants to the word-level. In *Proceedings of FMCAD*, pages 85–89. IEEE, 2007.
- [LB03] S. K. Lahiri and R. E. Bryant. Deductive verification of advanced out-of-order microprocessors. In *Proc. 15th International Conference on Computer-Aided Verification (CAV)*, volume 2725 of *LNCS*, pages 341–354, 2003.
- [LBGT04] S. K. Lahiri, R. E. Bryant, A. Goel, and M. Talupur. Revisiting positive equality. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 2988, pages 1–15, 2004.
- [LM05] S. K. Lahiri and M. Musuvathi. An efficient decision procedure for UTVPI constraints. In J. G. Carbonell and J. Siekmann, editors, *Proceedings of the 5th International Workshop on Frontiers of Combining Systems (FroCos '05)*, volume 3717 of *Lecture Notes in Artificial Intelligence*, pages 168–183. Springer, September 2005.
- [LMS04] I. Lynce and J. P. Marques-Silva. On computing minimum unsatisfiable cores. In *SAT*, 2004.
- [LRT⁺16] T. Liang, A. Reynolds, N. Tsiskaridze, C. Tinelli, C. Barrett, and M. Deters. An efficient SMT solver for string constraints. *Formal Methods in System Design*, 48(3):206–234, 2016.
- [LS04] S. K. Lahiri and S. A. Seshia. The UCLID decision procedure. In *Proc. 16th International Conference on Computer-Aided Verification (CAV)*, pages 475–478, July 2004.
- [Mat71] Y. V. Matiyasevich. Diophantine representation of recursively enumerable predicates. In J. E. Fenstad, editor, *Second Scandinavian Logic Symposium*, volume 63 of *Studies in Logic and the Foundations of Mathematics*, pages 171–177. North-Holland Publishing Company, 1971.
- [McM02] K. McMillan. Applying SAT Methods in Unbounded Symbolic Model Checking. In *Proc. CAV '02*, number 2404 in LNCS. Springer, 2002.
- [McM03] K. McMillan. Interpolation and SAT-based model checking. In *Proc. CAV*, 2003.
- [McM05] K. L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1), 2005.

- [MJ04] F. Maric and P. Janjic. ARGO-Lib: A generic platform for decision procedures. In *Proceedings of IJCAR '04*, volume 3097 of *Lecture Notes in Artificial Intelligence*, pages 213–217. Springer, 2004.
- [ML06] M. Moskała and J. Lopuszański. Fast quantifier reasoning with lazy proof explication. Technical report, Institute of Computer Science, University of Wrocław, May 2006.
- [MLA⁺05] M. N. Mneimneh, I. Lynce, Z. S. Andraus, J. P. Marques-Silva, and K. A. Sakallah. A Branch-and-Bound Algorithm for Extracting Smallest Minimal Unsatisfiable Formulas. In *Proc. SAT'05*, volume 3569 of *LNCS*. Springer, 2005.
- [MMZ⁺01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, 2001.
- [Mö197] M. O. Möller. *Solving Bit-Vector Equations – a Decision Procedure for Hardware Verification*. PhD thesis, University of Ulm, 1997.
- [MPP15] P. Manolios, J. Pais, and V. Papavasileiou. The Inez mathematical programming modulo theories framework. In *27th International Conference on Computer-Aided Verification (CAV)*, pages 53–69, 2015.
- [MZ03a] Z. Manna and C. Zarba. Combining decision procedures. In *Formal Methods at the Crossroads: from Panacea to Foundational Support*, volume 2787 of *Lecture Notes in Computer Science*, pages 381–422. Springer, November 2003.
- [MZ03b] Z. Manna and C. G. Zarba. Combining decision procedures. In *Formal Methods at the Cross Roads: From Panacea to Foundational Support*, volume 2757 of *Lecture Notes in Computer Science*, pages 381–422. Springer, 2003.
- [NO79] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. on Programming Languages and Systems*, 1(2):245–257, October 1979.
- [NO80] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the Association for Computing Machinery*, 27(2):356–364, April 1980.
- [NO03] R. Nieuwenhuis and A. Oliveras. Congruence closure with integer offsets. In *In 10th Int. Conf. Logic for Programming, Artif. Intell. and Reasoning (LPAR)*, volume 2850 of *LNAI*, pages 78–90. Springer, 2003.
- [NO05a] R. Nieuwenhuis and A. Oliveras. Decision Procedures for SAT, SAT Modulo Theories and Beyond. The BarcelogicTools. (Invited Paper). In G. Sutcliffe and A. Voronkov, editors, *12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR'05*, volume 3835 of *Lecture Notes in Computer Science*, pages 23–46. Springer, 2005.
- [NO05b] R. Nieuwenhuis and A. Oliveras. Proof-Producing Congruence Closure. In *Proceedings of the 16th International Conference on Term Rewriting and Applications, RTA'05*, volume 3467 of *LNCS*. Springer, 2005.
- [NO05c] R. Nieuwenhuis and A. Oliveras. DPLL(T) with exhaustive theory

- propagation and its application to difference logic. In K. Etessami and S. K. Rajamani, editors, *Proceedings of the 17th International Conference on Computer Aided Verification (CAV '05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 321–334. Springer, July 2005.
- [NOT05] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Abstract DPLL and abstract DPLL modulo theories. In F. Baader and A. Voronkov, editors, *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'04), Montevideo, Uruguay*, volume 3452 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2005.
- [NOT06] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, November 2006.
- [NPR⁺18] A. Niemetz, M. Preiner, A. Reynolds, C. Barrett, and C. Tinelli. Solving quantified bit-vectors using invertibility conditions. In *In Proceedings of the 30th International Conference on Computer-Aided Verification (CAV)*, pages 236–255, 2018.
- [NPSSV10] P. Nuzzo, A. Puggelli, S. A. Seshia, and A. L. Sangiovanni-Vincentelli. CalCS: SMT solving for non-linear convex constraints. In *Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 71–79, October 2010.
- [NR01] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 371–443. Elsevier and MIT Press, 2001.
- [OMA⁺04] Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov. Amuse: A Minimally-Unsatisfiable Subformula Extractor. In *Proc. DAC'04*. ACM/IEEE, 2004.
- [Opp80a] D. C. Oppen. Reasoning about recursively defined data structures. *Journal of the Association for Computing Machinery*, 27(3):403–411, July 1980.
- [Opp80b] D. C. Oppen. Complexity, convexity and combinations of theories. *Theoretical Computer Science*, 12:291–302, 1980.
- [Pap81] C. H. Papadimitriou. On the complexity of integer programming. *Journal of the Association for Computing Machinery*, 28(4):765–768, 1981.
- [PNB15] M. Preiner, A. Niemetz, and A. Biere. Better lemmas with lambda extraction. In R. Kaivola and T. Wahl, editors, *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015.*, pages 128–135. IEEE, 2015.
- [PNB17] M. Preiner, A. Niemetz, and A. Biere. Counterexample-guided model synthesis. In *TACAS (1)*, volume 10205 of *Lecture Notes in Computer Science*, pages 264–280, 2017.
- [Pra77] V. Pratt. Two easy theories whose combination is hard. Technical report, Massachusetts Institute of Technology, 1977. Cambridge,

- MA.
- [PRSS99] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small domains instantiations. In *Proceedings of the 11th International Conference on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 455–469. Springer, 1999.
- [PRSS02] A. Pnueli, Y. Rodeh, M. Siegel, and O. Strichman. The small model property: How small can it be? *Journal of Information and Computation*, 178(1):279–293, October 2002.
- [PS98] P. F. Patel-Schneider. DLP system description. In *Proc. DL-98*, pages 87–89, 1998.
- [Pud97] P. Pudlák. Lower bounds for resolution and cutting planes proofs and monotone computations. *J. of Symbolic Logic*, 62(3), 1997.
- [Pug91] W. Pugh. The omega test: A fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.
- [Rin96] C. Ringeissen. Cooperation of decision procedures for the satisfiability problem. In F. Baader and K. Schulz, editors, *Frontiers of Combining Systems: Proceedings of the 1st International Workshop, Munich (Germany)*, Applied Logic, pages 121–140. Kluwer Academic Publishers, March 1996.
- [RRZ05] S. Ranise, C. Ringeissen, and C. G. Zarba. Combining data structures with nonstably infinite theories using many-sorted logic. In B. Gramlich, editor, *Proceedings of the Workshop on Frontiers of Combining Systems*, volume 3717 of *Lecture Notes in Computer Science*, pages 48–64. Springer, 2005.
- [RS01] H. Ruess and N. Shankar. Deconstructing Shostak. In *16th Annual IEEE Symposium on Logic in Computer Science*, pages 19–28. IEEE Computer Society, June 2001. Boston, MA.
- [RSS07] A. Rybalchenko and V. Sofronie-Stokkermans. Constraint Solving for Interpolation. In *VMCAI, LNCS*. Springer, 2007.
- [SB05] S. A. Seshia and R. E. Bryant. Deciding quantifier-free Presburger formulas using parameterized solution bounds. *Logical Methods in Computer Science*, 1(2):1–26, December 2005.
- [SBD02] A. Stump, C. W. Barrett, and D. L. Dill. CVC: A cooperating validity checker. In E. Brinksma and K. G. Larsen, editors, *Proceedings of the 14th International Conference on Computer Aided Verification (CAV '02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 500–504. Springer, July 2002.
- [SDBL01] A. Stump, D. L. Dill, C. W. Barrett, and J. Levitt. A decision procedure for an extensional theory of arrays. In *Proceedings of the 16th IEEE Symposium on Logic in Computer Science (LICS '01)*, pages 29–37. IEEE Computer Society, June 2001.
- [Seb07] R. Sebastiani. Lazy Satisfiability Modulo Theories. *Journal on Satisfiability, Boolean Modeling and Computation – JSAT.*, 3, 2007.
- [Ses05] S. A. Seshia. *Adaptive Eager Boolean Encoding for Arithmetic Reasoning in Verification*. PhD thesis, Carnegie Mellon University, 2005.

- [Ses15] S. A. Seshia. Combining induction, deduction, and structure for verification and synthesis. *Proceedings of the IEEE*, 103(11):2036–2051, 2015.
- [Sho78] R. E. Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21(7), 1978.
- [Sho79] R. E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM*, 26(2):351–360, April 1979.
- [Sho84] R. Shostak. Deciding combinations of theories. *Journal of the Association for Computing Machinery*, 31(1):1–12, 1984.
- [SLB03] S. A. Seshia, S. K. Lahiri, and R. E. Bryant. A hybrid SAT-based decision procedure for separation logic with uninterpreted functions. In *40th Design Automation Conference (DAC '03)*, pages 425–430, June 2003.
- [SNS+18] Y. Shoukry, P. Nuzzo, A. L. Sangiovanni-Vincentelli, S. A. Seshia, G. J. Pappas, and P. Tabuada. SMC: satisfiability modulo convex programming. *Proceedings of the IEEE*, 106(9):1655–1679, 2018.
- [SNSV+17] Y. Shoukry, P. Nuzzo, A. Sangiovanni-Vincentelli, S. A. Seshia, G. J. Pappas, and P. Tabuada. SMC: Satisfiability modulo convex optimization. In *Proceedings of the 10th International Conference on Hybrid Systems: Computation and Control (HSCC)*, pages 19–28, April 2017.
- [SR02] N. Shankar and H. Rueß. Combining Shostak theories. In S. Tison, editor, *Int'l Conf. Rewriting Techniques and Applications (RTA '02)*, volume 2378 of *LNCS*, pages 1–18. Springer, 2002.
- [SS05] H. M. Sheini and K. A. Sakallah. A scalable method for solving satisfiability of integer linear arithmetic logic. In F. Bacchus and T. Walsh, editors, *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT '05)*, volume 3569 of *Lecture Notes in Computer Science*, pages 241–256. Springer, June 2005.
- [SS06] H. M. Sheini and K. A. Sakallah. A Progressive Simplifier for Satisfiability Modulo Theories. In *Proc. SAT'06*, volume 4121 of *LNCS*. Springer, 2006.
- [SSB02] O. Strichman, S. A. Seshia, and R. E. Bryant. Deciding separation formulas with SAT. In E. Brinksma and K. G. Larsen, editors, *Proc. 14th Intl. Conference on Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 209–222. Springer-Verlag, July 2002.
- [SSB07] S. A. Seshia, K. Subramani, and R. E. Bryant. On solving boolean combinations of UTVPI constraints. *Journal of Satisfiability, Boolean Modeling, and Computation (JSAT)*, 3:67–90, 2007.
- [ST17] R. Sebastiani and P. Trentin. On optimization modulo theories, maxsmt and sorting networks. In *In 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 231–248, 2017.
- [Str02a] O. Strichman. On solving Presburger and linear arithmetic with SAT. In *Formal Methods in Computer-Aided Design (FMCAD '02)*, LNCS 2517, pages 160–170. Springer-Verlag, November 2002.

- [Str02b] O. Strichman. Optimizations in decision procedures for propositional linear inequalities. Technical Report CMU-CS-02-133, Carnegie Mellon University, 2002.
- [TH96] C. Tinelli and M. T. Harandi. A new correctness proof of the Nelson–Oppen combination procedure. In F. Baader and K. Schulz, editors, *Frontiers of Combining Systems: Proceedings of the 1st International Workshop (Munich, Germany)*, Applied Logic, pages 103–120. Kluwer Academic Publishers, March 1996.
- [TR03] C. Tinelli and C. Ringeissen. Unions of non-disjoint theories and combinations of satisfiability procedures. *Theoretical Computer Science*, 290(1):291–353, January 2003.
- [TSSP04] M. Talupur, N. Sinha, O. Strichman, and A. Pnueli. Range allocation for separation logic. In *Proc. Computer-Aided Verification (CAV)*, volume 3114 of *Lecture Notes in Computer Science*, pages 148–161, 2004.
- [TZ04] C. Tinelli and C. Zarba. Combining decision procedures for sorted theories. In J. Alferes and J. Leite, editors, *Proceedings of the 9th European Conference on Logic in Artificial Intelligence (JELIA'04), Lisbon, Portugal*, volume 3229 of *Lecture Notes in Artificial Intelligence*, pages 641–653. Springer, 2004.
- [TZ05] C. Tinelli and C. Zarba. Combining nonstably infinite theories. *Journal of Automated Reasoning*, 34(3):209–238, April 2005.
- [vzGS78] J. von zur Gathen and M. Sieveking. A bound on solutions of linear integer equalities and inequalities. *Proceedings of the American Mathematical Society*, 72(1):155–158, October 1978.
- [WGG06] C. Wang, A. Gupta, and M. Ganai. Predicate learning and selective theory deduction for a difference logic solver. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*. ACM Press, 2006.
- [WW99] S. Wolfman and D. Weld. The LPSAT Engine & its Application to Resource Planning. In *Proc. IJCAI*, 1999.
- [YM05] G. Yorsh and M. Musuvathi. A combination method for generating interpolants. In *CADE*, volume 3632 of *LNCS*. Springer, 2005.
- [YM06] Y. Yu and S. Malik. Lemma Learning in SMT on Linear Constraints. In *Proc. SAT'06*, volume 4121 of *LNCS*. Springer, 2006.
- [ZLS06] J. Zhang, S. Li, and S. Shen. Extracting Minimum Unsatisfiable Cores with a Greedy Genetic Algorithm. In *Proc. AICAD*, volume 4304 of *LNCS*. Springer, 2006.
- [ZM02] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *Proc. CAV'02*, number 2404 in *LNCS*, pages 17–36. Springer, 2002.
- [ZM03] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. In *Proc. of SAT*, 2003.
- [ZMMM01] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.

This page intentionally left blank

Chapter 34

Stochastic Boolean Satisfiability

Stephen M. Majercik

34.1. Introduction

Stochastic satisfiability (SSAT) is an extension of satisfiability (SAT) that merges two important areas of artificial intelligence: logic and probabilistic reasoning. SSAT was initially suggested by Papadimitriou, who called it a “game against nature” [Pap85] and explored further by Littman, Majercik & Pitassi [LMP01]. SSAT is interesting both from a theoretical perspective—it is PSPACE-complete, an important complexity class—and from a practical perspective—a broad class of probabilistic planning problems can be encoded and solved as SSAT instances. In this chapter, we describe SSAT and its variants, their computational complexity, applications, analytical results, algorithms and empirical results, and related work. We conclude with a discussion of directions for future work.

34.2. Definitions and Notation

A stochastic satisfiability (SSAT) problem $\Phi = Q_1v_1 \dots Q_nv_n\phi$ is specified by:

1. a *prefix* $Q_1v_1 \dots Q_nv_n$ that orders a set of n Boolean variables $V = \{v_1, \dots, v_n\}$ and specifies the quantifier Q_i associated with each variable v_i , and
2. a *matrix* ϕ that is a Boolean formula constructed from these variables.

More specifically, the prefix $Q_1v_1 \dots Q_nv_n$ associates a quantifier Q_i , either existential (\exists) or “randomized” (\exists^{π_i}), with the variable v_i . An existentially (randomly) quantified variable is an *existential (randomized) variable*. A *sub-block* of variables in the prefix is any sequence of adjacent, similarly quantified variables; a *block* is a maximal sub-block. A block of existential (randomized) variables is an *existential (randomized) block*. We will use v_1, v_2, \dots for variables in general, x_1, x_2, \dots for existential variables, and y_1, y_2, \dots for randomized variables. An existential variable x_i is a standard Boolean variable, whereas a randomized variable y_i is a Boolean variable that has the value **true** with an arbitrary rational probability π_i . (A more restricted version of SSAT, in which all randomized variables are **true** with probability $1/2$, is defined in [LMP01].) $X \subseteq V$ is the set

of all existential variables in Φ and $Y \subseteq V$ is the set of all randomized variables in Φ . X_i is the subset of variables in the i^{th} existential block in the prefix of Φ and Y_i is the subset of variables in i^{th} randomized block in the prefix of Φ .

The matrix ϕ is assumed to be in conjunctive normal form (CNF), i.e. a set of m conjoined clauses, where each clause is a set of distinct disjoined literals. A *literal* l is either a variable v (a *positive* literal) or its negation \bar{v} (a *negative* literal). For a literal l , $|l|$ is the variable v underlying that literal and \bar{l} is the “opposite” of l , i.e. if l is v , \bar{l} is \bar{v} ; if l is \bar{v} , \bar{l} is v ; A literal l is **true** if it is positive and $|l|$ has the value **true**, or if it is negative and $|l|$ has the value **false**. A literal l is *existential* (*randomized*) if $|l|$ is existential (randomized). The probability that a randomized variable v has the value **true** (**false**) is denoted $\Pr[v]$ ($\Pr[\bar{v}]$). The probability that a randomized literal l is **true** is denoted $\Pr[l]$. As in a SAT problem, a clause is satisfied if at least one literal is **true**, and unsatisfied, or *empty*, if all its literals are **false**. The formula is satisfied if all its clauses are satisfied.

A partial assignment α of the variables V is a sequence of $k \leq n$ literals $l_1; l_2; \dots; l_k$ such that no two literals in α have the same underlying variable. Given l_i and l_j in an assignment α , $i < j$ implies that the assignment to $|l_i|$ was made before the assignment to $|l_j|$. A positive (negative) literal v (\bar{v}) in an assignment α indicates that the variable v has the value **true** (**false**). The notation $\Phi(\alpha)$ denotes the SSAT problem Φ' remaining when the partial assignment α has been applied to Φ ; that is:

1. clauses with **true** literals have been removed from the matrix,
2. **false** literals have been removed from the remaining clauses in the matrix,
3. all variables and associated quantifiers not in the remaining clauses have been removed from the prefix, and
4. the remaining variables and quantifiers in the prefix have been renumbered such that the index of the leftmost variable and its quantifier is 1 and the index of every other variable and quantifier is one more than the index of the immediately preceding variable and quantifier.

The notation $\phi(\alpha)$ denotes ϕ' , the matrix remaining when α has been applied. Similarly, given a set of literals L , such that no two literals in L have the same underlying variable, the notation $\Phi(L)$ denotes the SSAT problem Φ' remaining when the assignments indicated by the literals in L have been applied to Φ . And $\phi(L)$ denotes ϕ' , the matrix remaining when the assignments indicated by the literals in L have been applied. A literal $l \notin \alpha$ is *active* if some clause in $\phi(\alpha)$ contains l ; otherwise it is *inactive*.

Since the values of existential variables can be made contingent on the values of existentially and randomized variables that appear earlier in the prefix, the solution of an SSAT instance Φ takes the form of an *assignment tree* that specifies an assignment to each existential variable x_i for each possible instantiation of the randomized variables that precede x_i in the prefix. An optimal assignment tree, or *solution tree*, is one that yields the maximum probability of satisfaction of Φ . (A *decision* version of SSAT, which asks whether the probability of satisfaction exceeds a rational threshold $0 \leq \theta \leq 1$, is defined in [LMP01].) Formally, given an SSAT problem Φ , the maximum probability of satisfaction of Φ , denoted $\Pr^*[\Phi]$,

is defined according to the following recursive rules:

1. If ϕ contains an empty clause, $\text{Pr}^*[\Phi] = 0$.
2. If ϕ is the empty set of clauses, $\text{Pr}^*[\Phi] = 1$.
3. If the leftmost quantifier in the prefix of Φ is existential and the variable thus quantified is v , then $\text{Pr}^*[\Phi] = \max(\text{Pr}[\Phi(v)], \text{Pr}[\Phi(\bar{v})])$.
4. If the leftmost quantifier in ϕ is randomized and the variable thus quantified is v , then $\text{Pr}^*[\Phi] = (\text{Pr}[\Phi(v)] \times \text{Pr}[v]) + (\text{Pr}[\Phi(\bar{v})] \times \text{Pr}[\bar{v}])$.

34.2.1. Special Cases and Extensions of SSAT

Two special cases of SSAT, arrived at by placing restrictions on the type or ordering of the quantifiers, are of special interest. If all the variables are randomized, then Φ is a MAJSAT problem. (Papadimitriou [Pap85] referred to this as “Threshold SAT.”) Since none of the variables are existentially qualified, there is no optimal tree of existential variable assignments; the solution to a MAJSAT problem Φ is just $\text{Pr}^*[\Phi]$. If the prefix is composed of a single existential block followed by a single randomized block, then Φ is an E-MAJSAT problem. The solution to such a problem is an assignment α to X , the set of existential variables that maximizes $\text{Pr}[\Phi(\alpha)]$, where $\Phi(\alpha)$ is the MAJSAT problem that results when all the existential variables have been eliminated by applying α to Φ . Two other variants of SSAT are defined in [LMP01]:

- Alternating SSAT (ASSAT), a more constrained version of SSAT in which existential and randomized quantifiers strictly alternate, and
- Extended SSAT (XSSAT), called Extended SSAT in [LMP01], a further generalization of SSAT that includes universal quantifiers and encompasses SAT, QBF, SSAT, MAJSAT, and E-MAJSAT.

An XSSAT problem is essentially an SSAT problem in which variables can be universally quantified (\forall) as well as existentially and randomly quantified. A universally quantified variable is a *universal variable*. A block of universal variables is a *universal block*. We will use z_1, z_2, \dots for universal variables. $Z \subseteq V$ is the set of all universal variables in Φ . Z_i is the i^{th} universal block in the prefix of Φ .

The solution of an XSSAT instance Φ is an assignment of truth values to the existentially quantified variables that yields the maximum probability of satisfaction of Φ in the face of the demands of the universally quantified variables. If the leftmost variable in the prefix of XSSAT instance Φ is universally quantified ($Q_1 = \forall$), then $\text{Pr}^*[\Phi]$ must be such that the probability of satisfaction is attainable both when that variable is assigned the value **true** and when it is assigned the value **false**. Thus, $\text{Pr}^*[\Phi]$ for XSSAT instance Φ is defined according to the recursive rules stated above for an SSAT instance plus the following rule:

5. If the leftmost quantifier in the prefix of Φ is universal and the variable thus quantified is v , then $\text{Pr}^*[\Phi] = \min(\text{Pr}[\Phi(v)], \text{Pr}[\Phi(\bar{v})])$.

The decision version of XSSAT, which asks whether the probability of satisfaction exceeds a rational threshold $0 \leq \theta \leq 1$, is defined in [LMP01].) As noted in [LMP01], when an XSSAT instance Φ contains only existential and universal

quantifiers, the definition of $\text{Pr}^*[\Phi]$ is equivalent to the definition of satisfiability of a QBF instance. In fact, a QBF instance can be solved by replacing the universal quantifiers with randomized quantifiers whose associated probabilities are all strictly between 0 and 1, and checking whether the probability of satisfaction of the resulting SSAT instance is 1.

34.3. Complexity of SSAT and Related Problems

We briefly summarize some complexity results for SSAT; details can be found in [LMP01, Lit97, LGM98]. Each of SAT, MAJSAT, E-MAJSAT, SSAT, and QBF is complete for a particular complexity class, i.e. is “as hard as” any other problem in that class. A brief look at the satisfiability problems and the classes for which they are complete indicates the additional difficulty imposed by the introduction of randomized and universal quantifiers into the basic SAT problem. It also shows how these satisfiability problems relate to some standard propositional, probabilistic planning problems and belief network problems.

SAT is the prototypical NP-complete problem. An intuitive way to think about NP is that it is the class of decision problems for which one can guess and check an answer in time that is polynomial in the size of the problem. The problem of finding the most probable explanation (MPE) in a belief network [Dec96] is also NP-complete and, therefore, polynomially equivalent to SAT. A related problem from planning under uncertainty is determining whether there is some choice of actions such that the probability of the most likely trajectory through state space to the goal exceeds a given probability threshold.

MAJSAT is complete for the class PP (probabilistic polynomial time). By way of analogy with the earlier description of NP, one can think of PP as the class of decision problems for which one can guess and check an answer in time that is polynomial in the size of the problem and, if the correct answer is “yes,” the guessed answer will be correct with probability at least $1/2$. The problem of belief network inference is #P-complete [Rot96]. (roughly speaking, #P actually counts the number of satisfying assignments; PP just decides whether the probability of satisfaction is at least $1/2$.) In addition, any belief network (with rational conditional probability tables) can be represented as a Boolean formula. Plan evaluation in a probabilistic domain is a PP-complete planning problem [LGM98].

E-MAJSAT is NP^{PP} -complete [LGM98]. NP^{PP} is the class of decision problems for which one can guess a solution in time polynomial in the size of the problem (NP) and then perform a PP calculation to determine whether that answer is correct. The belief network problems of calculating a maximum a posteriori (MAP) hypothesis or a maximum expected utility (MEU) solution [Dec96] are NP^{PP} -complete. Finding optimal size-bounded plans in uncertain domains is also NP^{PP} -complete [LGM98]. One can think of the existential variables as corresponding to the hypothesis or plan and the randomized variables as corresponding to the uncertainty.

Finally, SSAT, ASSAT, and *XSSAT*, like QBF, are PSPACE-complete. PSPACE is the set of problems that can be solved using an amount of space that is polynomial in the size of the input. Solving a finite-horizon partially observable Markov decision process (POMDP) is also PSPACE-complete [PT87], when the

domain is specified compactly via probabilistic STRIPS operators or an equivalent representation [MGLA97], even if the domain is “fully observable” [LGM98]. Influence diagrams [Sha86] are a belief-network-like representation for the same type of problem.

34.4. Applications

The complexity results in Section 34.3 show that various planning problems can be efficiently transformed into a type of satisfiability problem (in time that is polynomial in the size of the problem). In fact, the most well-explored application of stochastic satisfiability is an extension of the deterministic-planning-as-satisfiability paradigm. (See Part 2, Chapter 19 for further discussion of planning and satisfiability.)

34.4.1. Planning

Three sound and complete probabilistic planners have been developed that solve a planning problem by converting the planning problem into a stochastic satisfiability problem and solving that problem instead. The stochastic satisfiability problem is designed so that the tree of existential variable assignments that yields $\Pr^* [\Phi]$ directly translates into the plan that has the highest probability of reaching the planning agent’s goal. All of these planners solve goal-oriented, finite-horizon, propositional probabilistic planning problems. MAXPLAN solves conformant planning problems by solving an E-MAJSAT encoding of the problem [ML98a]. ZANDER solves partially observable, contingent planning problems (partially observable Markov decision processes) by solving an SSAT encoding of the planning problem [ML03]. In Section 34.6.3.1, we describe these SSAT plan encodings and the technique ZANDER uses to solve them. Evidence that ZANDER is a viable alternative to the POMDP, partial order, and planning graph approaches to probabilistic planning is presented in [ML03]. Finally, DC-SSAT is a divide-and-conquer approach that takes advantage of the structure in the SSAT encoding of a completely observable planning problem to solve such planning problems 2-3 orders of magnitude faster than ZANDER using 2-3 orders of magnitude less space [MB05]. DC-SSAT is described in Section 34.6.3.2.

34.4.2. Belief Networks

[Rot96] showed that the problem of belief net inferencing can be reduced to MAJSAT, a type of SSAT problem. Thus, in principle, an SSAT solver could solve the inferencing problem by solving the MAJSAT encoding of that problem. That this approach would be an efficient alternative to standard belief net inferencing algorithms is supported by work in [BDP03c, BDP03a, BDP03b] that describes a DPLL-based approach to solving belief net inferencing problems that provides the same time and space performance guarantees as state-of-the-art exact algorithms and, in some cases, achieves an exponential speedup over existing algorithms.

34.4.3. Trust Management

SSAT has also shown promise in the development of algorithms for *trust management* (TM) systems. TM systems are application-independent infrastructures that can enforce access control policies within and between organizations to protect sensitive resources from access by unauthorized agents. State-of-the-art TM systems have significant limitations, e.g. difficulty in expressing partial trust, and a lack of fault tolerance that can lead to interruptions of service. [FK03] have shown that stochastic satisfiability can form the basis of a TM system that addresses these limitations, and work is in progress to develop efficient solution techniques for the SSAT problems generated by such a system.

34.5. Analytical Results

The structural properties of SAT problems have been studied extensively. For example, it is well-known that increasing the clause-to-variable ratio in random k -SAT problems produces an “easy-hardest-hard” pattern [SML96]). This pattern becomes particularly pronounced as the number of variables increases. It is conjectured that there exists a constant α_k such that a random k -SAT formula drawn from $\mathcal{F}_m^{k,n}$ (the distribution of formulas obtained by selecting independently m clauses of size k over n variables) is almost certainly satisfiable if $m/n < \alpha_k$ (as n gets large), and almost certainly unsatisfiable if $m/n > \alpha_k$. (See Part 1, Chapter 10 for a discussion of random k -SAT.)

The additional complexity introduced by the ordering of variables and quantifiers in the general SSAT problem has made it difficult to generate comparable analytical results for these problems. While there are no analytical results regarding the threshold behavior of random instances of general SSAT problems, however, there are some results for MAJSAT. [Fri97] showed that random SAT has a sharp 0–1 threshold (although the 0–1 threshold value α_k may be a function of n) and the ideas in that proof are general enough to extend to MAJSAT, i.e. there exists a function $\alpha_k(n)$ such that for any $\epsilon > 0$, there exists $t > 1$ such that a random MAJSAT instance with parameters (k, m, n, t) , where $m = (\alpha_k(n) - \epsilon)n$, is positive with high probability, while a random formula with parameters (k, m, n, t) , where $m = (\alpha_k(n) + \epsilon)n$, is not a positive instance with high probability, and moreover has zero satisfying truth assignments with high probability [LMP01]. This implies a 0–1 threshold although it does not specify the location of it.

[LMP01] describe some additional results for MAJSAT instances drawn from $\mathcal{F}_m^{k,n}$. These results are based on the decision version of the MAJSAT problem and hold only for instances in which, for each variable y_i , $\Pr[y_i] = 1/2$. A parameter t specifies a threshold probability of $1/2^{n-t}$. When $m \geq (\log_{8/7} 2)(n - t) \approx 5.19(n - t)$, a random 3-MAJSAT instance with m clauses and n variables almost certainly has fewer than 2^t satisfying assignments, and empirical estimates of the 0-1 threshold confirm the form of this inequality [LMP01]. Work by [KMPS95, KKKS98] shows that the actual number of satisfying assignments for most k -SAT formulas will be much smaller than the expected number of satisfying assignments and [LMP01], using results from [KMPS95], conclude that

when the matrix ϕ in a MAJSAT instance Φ is satisfiable, Φ will almost certainly be a positive instance when $t \leq ne^{-3m/n}$.

With respect to a lower bound on m as a function of t , assume that almost certainly, there are l disjoint clauses, C_1, C_2, \dots, C_l in a random MAJSAT formula with parameters k, m, n, t . Let $k = 3$. There are 2^{3l} total assignments for the l disjoint clauses. There are 7 ways to make each clause **true** for 3-CNF formulas. Since the clauses are disjoint, there are 7^l ways to make all the clauses true, so the exact number of assignments that make one or more clauses **false** is $2^{3l} - 7^l$, which is between $2^{3(l-1)}$ and 2^{3l} . This gives a lower bound on m as a function of t , since l can be chosen as a function of m and n such that almost certainly a 3-SAT random formula with m clauses over n variables contains at least l variable-disjoint clauses.

34.6. Algorithms and Empirical Results

Current algorithms for stochastic satisfiability include:

- 1) systematic algorithms for random problems and structured problems,
- 2) approximation algorithms for random problems and structured problems,
and
- 3) non-systematic algorithms for random problems.

In Section 34.6.2, we describe two sound and complete algorithms for solving arbitrary XSSAT problems. In Section 34.6.3, we describe two sound and complete algorithms for structured SSAT problems (SSAT encodings of probabilistic planning problems). In Sections 34.6.4 and 34.6.5, we describe approximation algorithms for random MAJSAT problems, random SSAT problems, and SSAT encodings of probabilistic planning problems. Finally, in Section 34.6.6, we describe a stochastic local search algorithm.

34.6.1. Decision Trees

We will use the idea of a *decision tree* to describe some of the following SSAT algorithms. A decision tree T over variables v_1, \dots, v_n is a binary decision tree where each internal node in the tree represents a variable v_i , and the two outgoing edges are labeled with the two possible values of v_i : **true** and **false**. Each leaf is labeled with either 0 or 1. Any complete assignment α to the variables is consistent with exactly one path of T . A decision tree T over v_1, \dots, v_n represents a CNF formula ϕ if for every truth assignment α , the value of the leaf node on the path consistent with α is equal to 0 if α does not satisfy ϕ and 1 if α does satisfy ϕ . A decision tree T for ϕ is *complete* if it is a full binary tree of height n . A given ϕ will have many decision trees that can represent it; some may be very small while others (such as the complete tree) can have exponential size in n .

A *canonical* decision tree for an XSSAT instance Φ is a decision tree T where the ordering of the variables along paths in T is consistent with the quantifier order. (Note that although the variables in a canonical decision tree must follow the ordering of the variables in the prefix, variables occurring in a block can be permuted within the block, both in Φ and in T , without altering the value.) Given

a canonical tree T for an XSSAT instance Φ , $\text{Pr}^*[\Phi]$ can be computed from T by labeling the intermediate nodes of T with rational numbers in a bottom-up fashion. A leaf is labeled 0 (1) if the assignment specified by the path to that leaf is unsatisfying (satisfying). Then, if node w representing variable v_i is unlabeled, and has children w_0 and w_1 with labels $L(w_0)$, $L(w_1)$, respectively, then:

1. if $Q_i = \exists$, then $L(w) = \max(L(w_0), L(w_1))$,
2. if $Q_i = \forall$, then $L(w) = \min(L(w_0), L(w_1))$,
3. if $Q_i = \forall$, then $L(w) = (L(w_0) * \text{Pr}[\bar{v}] + L(w_1) * \text{Pr}[v])$,

and $\text{Pr}^*[\Phi]$ is the label of the root.

Note that while a decision tree ignores the quantifier ordering, a canonical decision tree follows it exactly.

34.6.2. Systematic Algorithms

Two sound and complete algorithms for solving arbitrary SSAT problems (problems with no special structure) were described in [LMP01]. The first is a modification of the Davis-Putnam-Logemann-Loveland (DPLL) algorithm for solving SAT instances [DLL62] (the basis of most modern SAT engines). This DPLL-based algorithm, described in Section 34.6.2.1, solves XSSAT problems (and, thus, SSAT, E-MAJSAT, and MAJSAT problems as well). The second algorithm is also a DPLL-based algorithm for XSSAT problems and is guaranteed to solve an XSSAT instance in time that is efficient in the size of the smallest *decision tree* (defined in Section 34.6.2.2). As discussed in Section 34.6.2.2, however, this algorithm does not provide the basis for an efficient XSSAT algorithm.

34.6.2.1. A Davis-Putnam-Logemann-Loveland Algorithm

The Davis-Putnam-Logemann-Loveland (DPLL) algorithm for Boolean satisfiability [DLL62] searches for a satisfying assignment to a SAT instance by enumerating partial assignments and taking advantage of opportunities to prune assignments that could not lead to a satisfying assignment. DPLL needs to deal with only the implicitly existentially quantified variables in a SAT instance, but the algorithm can be extended to handle the addition of randomized quantifiers in an SSAT instance, and randomized and universal quantifiers in an XSSAT instance. Pruning rules for existentially and universally quantified variables are described in [DLL62] and [CGS98], respectively. These rules, plus pruning rules for randomly quantified variables are the basis for *evalssat*, an algorithm for XSSAT problems presented in [LMP01]. Since this algorithm is the basis of all the algorithms that have been developed for stochastic satisfiability, we will describe it in some detail.

The *evalssat* algorithm computes $\text{Pr}^*[\Phi]$ for XSSAT instance Φ from the definition of $\text{Pr}^*[\Phi]$ in Section 34.2.1. Thus, the basis of this algorithm is an enumeration of all assignments by assigning values to the variables in the order in which they occur in the prefix. The algorithm attempts to avoid enumerating all assignments, however, by using techniques adapted from the basic DPLL algorithm—*unit propagation* and *pure variable elimination*—as well as a *threshold pruning* technique similar to branch-and-bound techniques and techniques for

solving AND-OR and MINIMAX trees [Nil80]. Existential variable nodes are analogous to OR, or MAX, nodes, and randomized variable nodes are analogous to AND, or MIN, nodes. However, the probabilities associated with randomized variables (the opponent is nature) make the analogy somewhat inexact. The assignment trees searched by *evalssat* are more similar to MINIMAX trees with randomized nodes [Bal83], i.e. a sequence of alternating moves by opposing players mediated by random events.

Figure 34.1 provides pseudocode for DPLL-XSSAT, a modification of *evalssat* to accommodate randomized variables with arbitrary rational probabilities. The algorithm takes SSAT instance Φ and low and high probability thresholds θ_{\min} and θ_{\max} . It returns a value less than θ_{\min} if and only if the value of the XSSAT formula is less than θ_{\min} , a value greater than θ_{\max} if and only if the value of the XSSAT formula is greater than θ_{\max} , and otherwise the exact value of the XSSAT formula. Thus, it can be used to solve the XSSAT decision problem by setting $\theta_{\min} = \theta_{\max} = \theta$. It can also be used to compute the exact value of the formula by setting $\theta_{\min} = 0$ and $\theta_{\max} = 1$. Existing implementations of this algorithm also construct and return τ_{Φ} , the optimal assignment tree, but the construction of this tree is straightforward and, for the sake of clarity, we omit the details of this tree construction in the algorithm description. A proof of the correctness of *evalssat*, from which the correctness of DPLL-XSSAT follows, is provided in [LMP01].

Unit Propagation: A literal l is *unit* if it is the only literal in some clause (and we will refer to such a clause as a *unit clause*). If l is universal, the entire formula is **false** since it will not be the case that all values of $|l|$ make the formula **true** and $\Pr^*[\Phi] = 0$. If l is existential or randomized, then setting $|l|$ such that l is **false** immediately produces an empty clause and $\Pr^*[\Phi] = 0$. In this case, the algorithm can disregard the ordering of the variables in the prefix and choose the forced value. If l is existential, $\Pr^*[\Phi] = \Pr[\Phi(l)]$; if l is randomized, $\Pr^*[\Phi] = \Pr[\Phi(l)] \times \Pr[l]$.

Pure Variable Elimination: A literal l is *pure* if l is active and \bar{l} is inactive. If l is existential or universal, the algorithm can disregard the ordering of the variables in the prefix and choose the value consistent with the “goal” of the quantifier. If l is existential, the algorithm maximizes the probability of satisfaction by setting the value of $|l|$ to agree with l . If l is universal, the algorithm minimizes the probability of satisfaction by setting the value of $|l|$ to disagree with l . If l is randomized, there is no comparable simplification. Setting a pure randomized variable contrary to its sign in the formula can still yield a probability of satisfaction greater than 0, and this must be taken into account when computing the maximum probability of satisfaction.

Threshold Pruning: Threshold pruning, a mechanism absent from the DPLL algorithm for SAT, can boost efficiency by sometimes making it unnecessary to compute both $\Pr[\bar{v}_i]$ and $\Pr[v_i]$ when branching on a variable v_i . Threshold pruning computes *low* and *high* thresholds for the probability of satisfaction at each node in the solution tree (θ_{\min} and θ_{\max} in Figure 34.1). These thresholds delimit an interval outside of which it is unnecessary to compute the exact value of the probability of satisfaction (so threshold pruning cannot be used to solve instances in which the exact probability of satisfaction is required). Given a par-

```

DPLL-XSSAT( $\Phi, \theta_{\min}, \theta_{\max}$ )
  if  $\phi$  contains an empty clause:
    return 0;
  if  $\phi$  is the empty set of clauses:
    return 1;

  // Unit Propagation
  if there is a unit literal  $l$  and  $|l| = v_i$ :
    if  $Q_i = \exists$ :
      return DPLL-XSSAT( $\Phi(l), \theta_{\min}, \theta_{\max}$ );
    if  $Q_i = \forall$ :
      return 0;
    if  $Q_i = \forall$ :
      return DPLL-XSSAT( $\Phi(l), \theta_{\min} / \Pr[l], \theta_{\max} / \Pr[l]$ )  $\times \Pr[l]$ ;

  // Pure Variable Elimination
  if there is a pure literal  $l$  and  $|l| = v_i$ :
    if  $Q_i = \exists$ :
      return DPLL-XSSAT( $\Phi(l), \theta_{\min}, \theta_{\max}$ );
    if  $Q_i = \forall$ :
      return DPLL-XSSAT( $\Phi(\bar{l}), \theta_{\min}, \theta_{\max}$ );

  // Branching
  if the left-most quantifier  $Q_1 = \exists$ :
     $p_0 = \text{DPLL-XSSAT}(\Phi(\bar{x}_1), \theta_{\min}, \theta_{\max})$ ;
    if  $p_0 \geq \theta_{\max}$ :
      return  $p_0$ ;
     $p_1 = \text{DPLL-XSSAT}(\Phi(x_1), \max(\theta_{\min}, p_0), \theta_{\max})$ ;
    return  $\max(p_0, p_1)$ ;

  if the left-most quantifier  $Q_1 = \forall$ :
     $p_0 = \text{DPLL-XSSAT}(\Phi(\bar{z}_1), \theta_{\min}, \theta_{\max})$ ;
    if  $p_0 < \theta_{\min}$ :
      return  $p_0$ ;
     $p_1 = \text{DPLL-XSSAT}(\Phi(z_1), \theta_{\min}, \min(p_0, \theta_{\max}))$ ;
    return  $\min(p_0, p_1)$ ;

  if the left-most quantifier  $Q_1 = \forall$ :
     $p_0 = \text{DPLL-XSSAT}(\Phi(\bar{y}_1), (\theta_{\min} - \Pr[y_1]) / \Pr[\bar{y}_1], \theta_{\max} / \Pr[\bar{y}_1])$ ;
    if  $p_0 \Pr[\bar{y}_1] + \Pr[y_1] < \theta_{\min}$ :
      return  $p_0 \Pr[\bar{y}_1]$ ;
    if  $p_0 \Pr[\bar{y}_1] \geq \theta_{\max}$ :
      return  $p_0 \Pr[\bar{y}_1]$ ;
     $p_1 = \text{DPLL-XSSAT}(\Phi(y_1), (\theta_{\min} - p_0 \Pr[\bar{y}_1]) / \Pr[y_1], (\theta_{\max} - p_0 \Pr[\bar{y}_1]) / \Pr[y_1])$ ;
    return  $p_0 \Pr[\bar{y}_1] + p_1 \Pr[y_1]$ ;

```

Figure 34.1. The DPLL-XSSAT algorithm generalizes the DPLL algorithm for satisfiability to solve XSSAT problems.

tial assignment α at a node, the solver can return the probability of satisfaction of the first branch explored (adjusted by the probability of that branch occurring if it is a randomized variable node) if that probability of satisfaction allows the solver to determine that $\Pr[\Phi(\alpha)]$ will fall outside the range $[\theta_{\min}, \theta_{\max}]$. In other words, the solver can leave the other branch of a variable unexplored when it is clear that 1) even the highest probability of satisfaction obtainable by exploring that branch will still leave the low threshold unattained, or 2) the probability of satisfaction that would be obtained by exploring that branch is not necessary to attain the high threshold. Threshold pruning also allows the solver to prune some subtrees in the \bar{l} subtree based on the probability of satisfaction associated with the l subtree (e.g. if $|l|$ is an existential variable, $\Pr[\Phi(\alpha; l)] = 0.6$, and the solver begins exploring the \bar{l} subtree, it can quit as soon as it determines that $\Pr[\Phi(\alpha; \bar{l})] < 0.6$ and that, therefore, l will be the better choice). Thresholds need to be adjusted in certain cases as the current partial assignment is extended, particularly in the case of randomized variable assignments. These adjustments are indicated in Figure 34.1; further details are available in [LMP01].

In [LMP01], *evalssat* with unit propagation, pure variable elimination, threshold pruning, and branching based on strict quantifier ordering, was tested on a set of 1,000 randomly generated formulas with $n = 20$ variables, $k = 3$ literals per clause, and 141 clauses. In these experiments, the decision version of the problems was solved and the probability of satisfaction threshold was expressed as $\theta = 1/2^{n-t}$ for integer t in the range 0 to n , i.e. 2^t satisfying assignments are required. The number of recursive calls required was used as the measure of the work required to solve an instance. We summarize the results of some of these tests here. Details of these tests and other test results are available in [LMP01].

MAJSAT problems exhibit the same easy-hardest-hard pattern as SAT problems as the clause-to-variable ratio is increased, although the peak difficulty appears to be higher than the peak for comparable SAT problems and occurs at a lower clause-to-variable ratio. The peak increases as the threshold probability of satisfaction increases since the threshold pruning rules apply less often to randomized variables as the threshold probability increases; i.e. it is less likely that the probability of satisfaction obtained from exploring the first branch of a randomized variable will be sufficient to make exploring the other branch unnecessary. In unsatisfiable MAJSAT instances (probability of satisfaction is 0), however, a higher threshold probability of satisfaction increases the effectiveness of threshold pruning, since, given the probability of satisfaction of 0 obtained from the first branch, even a probability of satisfaction of 1 from the second branch would be insufficient to attain the required probability of satisfaction.

The higher level of difficulty of MAJSAT instances relative to SAT instances was explored further in a set of experiments with E-MAJSAT problems. In E-MAJSAT problems, a single block of existential variables is followed by a single block of randomized variables, so these problems allow one to explore a range of problem types between SAT (n existential variables and 0 randomized variables) and MAJSAT (0 existential variables and n randomized variables). These experiments suggested that work increased logarithmically as the proportion of existential variables decreases, a result that runs counter to complexity class considerations since E-MAJSAT is in higher complexity class than MAJSAT. These

results suggest that MAJSAT is empirically more difficult than E-MAJSAT which is, in turn, empirically more difficult than SAT.

[LMP01] also compared SAT, E-MAJSAT, ASSAT, and MAJSAT problems of the same size. (The E-MAJSAT instances had equal numbers of existential and randomized variables.) For each problem type, the threshold value was chosen so as to maximize the difficulty of the instances (as observed empirically). Again, all classes exhibited the easy-hardest-hard pattern of difficulty as the clause-to-variable ratio increased. The same relative difficulty was observed for MAJSAT, E-MAJSAT, and SAT; the difficulty of ASSAT was very close to that of E-MAJSAT, once again suggesting that difficulty increases with the number of randomized variables (since E-MAJSAT and ASSAT problems with the same number of variables have the same number of existential and randomized variables). This makes sense since pure variable elimination cannot be applied to randomized variables and the threshold pruning rules are weaker for randomized quantifiers. The impact of quantifier type on problem difficulty has also been noted in QBF problems [GW98, CGS97].

Branching Heuristics: When the DPLL algorithm exhausts opportunities to apply unit resolution and pure variable elimination, it must select an unassigned variable v from the current simplified formula to branch on, i.e. check for satisfying assignments both when v is assigned `true` and when v is assigned `false`. The order in which variables are selected is critical—an appropriate branching heuristic can reduce the running time of the DPLL algorithm on SAT instances by several orders of magnitude—and there has been a great deal of research into efficient branching heuristics for SAT. (See Part 1, Chapter 8 for a discussion of branching heuristics for satisfiability.) [LMP01] showed that these SAT branching heuristics can be adapted for SSAT problems, but branching in these problems is more restricted. To ensure that the value of the formula is computed correctly, branching heuristics for SSAT must choose a variable from the leftmost quantifier block. Hence, branching heuristics have no impact in ASSAT instances, since each block contains only a single variable. This suggests that branching heuristics will be more effective as the block size increases.

This conjecture was tested in [LMP01] using six branching heuristics adapted from SAT branching heuristics: 1) choose a variable randomly, 2) find the literal l that satisfies the most clauses in the current formula and chooses the variable $|l|$, 3) 2-sided Jeroslow-Wang [HV95], which chooses a variable whose literals appear in a large number of short clauses, 4) positive, 2-sided Jeroslow-Wang [HV95]), which is Jeroslow-Wang applied to the subset of variables that appear in at least one clause containing all positive literals, 5) choose the variable that has many occurrences in small clauses (MOMS) [JW90], and 6) a heuristic that prefers variables that will lead to a maximal number of unit resolutions [BS97, Fre95, CA96]. In general, all but the random heuristic and the Jeroslow-Wang heuristic decreased the size of the DPLL tree constructed (as compared to the size of the tree when the variable ordering in the prefix was followed strictly). Although there were some anomalies, this improvement was positively correlated with block size. MOMS and the heuristic that tries to satisfy the most clauses were particularly effective; the success of the latter, which is computationally much simpler than all but the random heuristic, is surprising. Branching heuristics that take advantage

of the structure in SSAT encodings of planning problems are described in [ML03], but similar tests have not been conducted for such problems.

Nonchronological Backtracking: Nonchronological backtracking (NCB) is an important technique in most state-of-the-art SAT engines and has also been used for structured QBF problems [GNT03]. (See Part 1, Chapter 4 for more on nonchronological backtracking in satisfiability and Part 2, Chapter 30 for more on nonchronological backtracking in quantified Boolean formulas.) An NCB-augmented solver attempts to avoid exploring both values of a variable whenever possible. It does this by computing a *reason* whenever the current partial assignment α satisfies the formula or leads to a contradiction. A reason R is a subset of literals in α that is responsible for that result. When backtracking, the solver can sometimes use this reason to recognize situations in which exploring the second value of a particular variable v is unnecessary.

In an SSAT instance, a reason R is created at the end of a path in the solution tree when the solver has established that, given the current partial assignment α , either $(Pr(\Phi(\alpha)) = 0)$ or $(Pr(\Phi(\alpha)) = 1)$. A literal l from α is included in the reason only if changing the value of the underlying variable $|l|$ could have an impact on the result obtained at that leaf. If the probability of satisfaction at a leaf is 1, the solver does not need to check the other value of existential variables in the assignment, but *does* need to check the other value of any randomized variable to see if it will lower the probability of satisfaction. If the probability of satisfaction at a leaf is 0, there will be a clause which has just become empty that is responsible for this. The solver needs to check the other value of any existential *or* randomized variable that was initially in this clause to see if the probability of satisfaction of 0 can be improved, either by making a better choice in the case of an existential variable, or by discovering that the other value of a randomized variable leads to a probability of satisfaction greater than 0. The absence of a literal l in a reason (which is modified as the solver backtracks) indicates that changing the value of $|l|$ will not affect the probability of satisfaction, allowing the algorithm to avoid exploring that value. Rules for creating and modifying reasons in SSAT problems are described in [Maj04].

An SSAT solver augmented with this NCB mechanism was tested on random problems generated using a fixed-clause k -SSAT model similar to the k -QBF Model A of Gent and Walsh [GW99]. While tests on some randomly generated problems indicated that NCB can improve efficiency greatly, decreasing the median number of nodes visited and the median solution time by as much as five orders of magnitude in some problems, the number and type of problems in the test set are too limited to draw any general conclusions. In fact, tests on a second set of problems indicated that NCB becomes less effective as $Pr^*[\Phi]$ decreases, due to an increase in the size of the reasons, thereby reducing their effectiveness in reducing search. Furthermore, NCB degraded performance in tests on structured problems (SSAT encodings of probabilistic planning problems from [ML03]). These results are in sharp contrast to the effectiveness of NCB in solving structured SAT problems [BS97] and structured QBF problems [GNT03] (although [GNT03] do report some instances in which the overhead associated with NCB outweighs its benefits). This poor performance on SSAT encodings of planning problems may be due to the nature of the encoding, and it is possible

that other types of encodings [MR02] would improve the effectiveness of NCB. It is also possible that NCB would be more effective in SSAT instances with other types of structure.

Memoization: Memoization, a dynamic programming technique that caches solved subproblems and their solutions, can greatly increase the speed of an SSAT solver [ML98a]. An SSAT solver using memoization saves every subproblem encountered during the solution process along with its probability of satisfaction, thereby allowing the solver to avoid needless exploration of identical subtrees. Both MAXPLAN [ML98a] and current versions of ZANDER created by the author rely heavily on *memoization*. While memoization is memory intensive, it allows a solver to handle larger SSAT encodings of planning problems. In [Maj07], results are shown for three types of memoization in ZANDER: no memoization, memoization of all subproblems, and memoization of only those subproblems that are solved when both values of a variable are explored (i.e. do not memoize a subproblem that arises when a unit clause forces the value of a variable). Results indicated that the last option is usually the best one. For example, for a planning problem in which the SSAT encodings contained $9t$ variables and $31t - 1$, where t is the number of time steps in the plan, ZANDER with no memoization could find the optimal plans for both a 5-step problem and a 10-step problem. Using full memoization allowed ZANDER to find optimal plans up to 15 steps, and selective memoization ZANDER to find optimal plans up to 20 steps. At the 25-step level all variants of ZANDER exhausted memory.

Caching can be used to reduce the memory requirements of memoization. This approach uses a fixed amount of memory as a cache for subproblems and their probabilities of satisfaction along with a replacement policy to decide what subproblem to evict when the cache is full and a new subproblem needs to be stored. Two standard cache replacement policies—first-in-first-out (FIFO) and least-recently-used (LRU)—were tested and compared in [ML98b]. LRU outperformed FIFO by an average of approximately 18% across a range of cache sizes for a small planning problem. An approach that saved only “mid-size” problems (since large subproblems are unlikely to be reused and small subproblems can be quickly resolved), was ineffective on that problem. An approach that saved subproblems based on difficulty (size of DPLL subtree), was moderately effective, yielding up to a 37% decrease in CPU seconds compared to straight LRU caching. The optimal difficulty range, however, was determined experimentally and tests with other domains indicated that the range was problem dependent. Furthermore, the benefits of this modified LRU caching technique disappeared as the size of the planning problem increased.

Memoization has also been found to be useful in the context of DPLL algorithms for SAT [BIPS06], #SAT [BDP03b, SBB⁺04], and QBF [GKM04]. #SAT, the problem of counting the number of satisfying assignments in a SAT instance, is #P-complete and is thus closely related to *MAJSAT*, which is PP-complete. [BIPS06] formalize and analyze the complexity of proof systems for *formula caching* DPLL SAT algorithms that save unsatisfiable subproblems. *Component caching*, in which disjoint components of a subproblem, rather than the subproblem, are cached, has been shown to be particularly effective. [BDP03b] showed that component caching is theoretically competitive with the best known

#SAT algorithms and [SBB⁺04] showed that these theoretical results translate into competitive algorithms. They showed that component caching and clause learning can be effectively integrated in an algorithm to solve #SAT problems. Their algorithm is faster than *relsat* [BS97, BP00] by up to several orders of magnitude.

34.6.2.2. A Universal Search Algorithm for XSSAT

The worst-case running time of DPLL-XSSAT is exponential in the size of the formula. Given an unsatisfiable SAT instance that is known to require an exponential size resolution proof, DPLL-XSSAT produces a resolution refutation, so resolution lower bounds for specific formulas give exponential time lower bounds on the running time of DPLL-XSSAT. For random formulas over a broad range, the decision tree complexity of the underlying 3-CNF formula and, therefore, the running time of DPLL-XSSAT (or any decision-tree based algorithm) without pruning, will be exponential: [LMP01], based on results in [BKPS98], show the following:

Theorem 1. Let ϕ be a 3-CNF formula chosen uniformly at random from $\mathcal{F}_m^{k,n}$. For all n sufficiently large, for all m , $n \leq m \leq n^{5/4}$, with high probability any decision tree for ϕ requires exponential size in n .

where $\mathcal{F}_m^{k,n}$ denotes the distribution of formulas obtained by selecting independently m clauses of size k over n variables. A clause is generated by randomly selecting one of the n variables k times, rejecting duplicate selections, and randomly negating the variable with probability $1/2$.

Furthermore, DPLL-XSSAT is not guaranteed to search for the smallest tree. There is an algorithm, however, that is guaranteed to find a close-to-optimal size tree efficiently [LMP01]. Let Φ be an instance of XSSAT with κ consecutive blocks of variables X^1, \dots, X^κ . As described in Section 34.6.1, $\text{Pr}^*[\Phi]$ can be calculated by labeling a canonical decision tree for Φ in a way that is consistent with the recursive definition of $\text{Pr}^*[\Phi]$ given in Section 34.2.

The DPLL-XSSAT algorithm described in Section 34.6.2.1, however, can be generalized to search for a decision tree that is more general than a canonical decision tree (branches need not follow the precise quantifier ordering), but has the following properties: for every i, j , $i < j$, every variable in the block X^i is branched upon in the decision tree before every variable in the block X^j , with the possible exception of variables that appear in unit clauses or pure variables. A decision tree for Φ with these properties is a DPLL *decision tree*, or DPLL tree, for Φ . Thus, a DPLL tree falls in between a decision tree and a canonical decision tree, in the sense that while a decision tree ignores the quantifier ordering and a canonical decision tree follows the ordering exactly, a DPLL tree ignores the quantifier ordering only within blocks. The evaluation of a DPLL tree for Φ is obtained by evaluating each vertex, starting at the leaves, in the manner described in Section 34.6.1 for canonical decision trees.

Based on the work of [CEI96, BP96, BSW99], which shows that there is a relatively efficient deterministic procedure for finding a small tree-like resolution proof for a SAT instance, the following theorem is proved in [LMP01]:

Theorem 2. There is a deterministic algorithm that takes as input an XSSAT instance Φ with m clauses and n underlying variables and outputs T , a DPLL decision tree for ϕ . Moreover, the running time of the algorithm is at most $n^{O(\log s)}O(m)$, where s is the size of the smallest DPLL decision tree for Φ .

Thus, there is an efficient (quasi-polynomial) deterministic procedure for finding a small DPLL tree. The algorithm operates by dovetailing computations for all possible decision trees in a recursive fashion, terminating the computation of suboptimal subtrees as soon as their non-optimality becomes known; details are available in [LMP01]. Although this universal search algorithm has superior theoretical properties to the DPLL algorithm of Section 34.6.2.1, it is possible that the overhead of the dovetailed recursive calls would dominate for most instances.

Since a general decision tree does not respect the quantifier ordering, for a given XSSAT instance Φ , there may, in fact, be a decision tree that is smaller than the DPLL tree. This would suggest that an algorithm that efficiently constructs such a tree would be desirable. [LMP01], however, state a result from [Uma99] showing that an arbitrary decision tree cannot be used to evaluate an XSSAT formula efficiently unless an unexpected collapse of complexity classes occurs. There are decision trees somewhat less restricted than DPLL trees that could be used to evaluate an XSSAT formula [LMP01], but efficient algorithms to find such trees are so far lacking.

34.6.3. Systematic Algorithms For Structured Problems

There are currently three SSAT algorithms that are designed to take advantage of the structure in a particular type of SSAT problem: MAXPLAN [ML98a], ZANDER [ML03], and DC-SSAT [MB05]. Since the algorithm used by ZANDER generalizes the one used by MAXPLAN, we will only describe the ZANDER algorithm here.

34.6.3.1. ZANDER

ZANDER solves goal-oriented, finite-horizon, partially observable, probabilistic propositional contingent planning problems by converting the planning problem into an SSAT instance and solving that problem instead [ML03]. ZANDER works on partially observable probabilistic propositional planning domains consisting of a finite set of distinct *state propositions*, each of which may be **true** or **false** at any discrete time t . A *state* is an assignment of truth values to these propositions. A possibly probabilistic *initial state* is specified by a set of decision trees, one for each proposition. *Goal states* are specified by a partial assignment to the set of propositions; any state that extends this partial assignment is a goal state. Each of a finite set of *actions* probabilistically transforms a state at time step t into a state at time step $t + 1$ and so induces a probability distribution over the set of all states at time step $t + 1$. A set of *random propositions* encodes the uncertainty in the agent's actions and its knowledge of its current state. A subset of the set of state propositions is designated *observable* and a set of *observation propositions* is created corresponding to these observable state propositions. (Note that when the planning problem assumes complete observability, a subset

of state propositions can be declared observable without creating a separate set of observation propositions.) The task is to find an action for each time step t as a function of the value of observation propositions at time steps before step t and that maximizes the probability of reaching a goal state.

Given a plan horizon T , ZANDER time-indexes each proposition and action so the planner can reason about what happens when. Variables are created to record the status of actions and state and random propositions in a T -step plan by taking four separate cross products: actions and time steps 1 through T (existential variables), state propositions and time steps 0 through T (existential variables), random propositions and time steps 1 through T (randomized variables), and observation propositions and time steps 1 through T (randomized variables). The total number of variables in the CNF formula is $V = (A + S + R + O)T + P$, where A , S , R , and O are the number of actions, state propositions, random propositions, and observation propositions, respectively. Using these variables, ZANDER translates the planning problem into an SSAT problem by creating unit clauses that enforce initial and goal conditions, clauses that ensure that exactly one action is taken at each time step, clauses that model the (possibly probabilistic) effects of actions on state propositions, and frame axioms clauses that model lack of effect of some actions on some state propositions unchanged. There are, currently, four ways to encode this type of planning problem as an SSAT instance [MR02]: three of them restrict the plan to linear actions—a single action at each time step of the plan—whereas the fourth encoding allows parallel actions—more than one action at a given time step. The three linear-action encodings differ in the way they encode frame axioms: classical frame axioms, simple explanatory frame axioms, and complex explanatory frame axioms. These encodings, with the exception of the linear-action encoding with complex explanatory frame axioms, are similar to the encodings described in [KMS96] for deterministic planning as satisfiability. Empirically, linear-action encodings with simple explanatory frame axioms and parallel-action encodings are the most efficient, yielding solution times as much as three orders of magnitude faster than the linear-action encoding with classical frame axioms.

Essentially, ZANDER uses the SSAT solver described in Section 34.6.2.1, with some modifications, to find the assignment tree that specifies the assignments to existentially quantified action variables for all possible settings of the observation variables, such that the probability of satisfaction (which is also the probability that the plan will reach the goal) is maximized [ML03]. There are three important modifications:

1. Empirical results indicate that pure variable elimination does not provide significant improvement when solving SSAT encodings of planning problems—frequently, when a variable becomes pure, it also becomes unit and is eliminated through unit propagation—so ZANDER does not use this heuristic [ML03].
2. Randomized variables representing observations are intended to mark possible branch points in the plan, not to encode the probability of making that observation, which is encoded by a separate randomized variable. The algorithm needs to sum the probabilities of satisfaction over both branches of an observation variable. In [ML03], this is accomplished by associating

a probability of $1/2$ with each randomized variable representing an observation and adjusting the final probability of success upward by a factor of 2 for each observation variable. Current versions of ZANDER created by the author, however, sum the probabilities at each observation node directly.

3. The branching heuristic follows the time indexing of the variables when there are variables with different time indices in the same block, branching on variables with a lower time index first. In blocks in which all variables have the same time index (e.g. action variable blocks and observation variable blocks), however, ZANDER follows the arbitrary ordering imposed during the creation of the SSAT encoding of the planning problem.

Results reporting ZANDER's performance on a suite of test problems compared to seven other probabilistic planners are reported in [ML03].

34.6.3.2. DC-SSAT

The most significant advance in SSAT solvers since the DPLL-based algorithms described in Sections 34.6.2.1 and 34.6.3.1 is DC-SSAT, a sound and complete SSAT solver that uses a divide-and-conquer approach to take advantage of the structure in a COPP-SSAT problem, i.e. an SSAT encoding of a completely observable planning problem [MB05]. In such a problem, the agent has complete and accurate information about its environment, although the effects of its actions may be uncertain.

The key idea underlying DC-SSAT is to enlarge the size and scope of the units being manipulated by the solver. In order to solve an SSAT problem, any solver needs to consider, directly or indirectly, every possible satisfying assignment. ZANDER does this by exploring possible values for individual variables, DC-SSAT takes advantage the linear structure in a COPP-SSAT problem—variables in an existential block appear in clauses only with variables from the same existential block and variables from adjacent randomized blocks—to define an equivalence relation on the existential variables that partitions the SSAT problem into subproblems that are relatively independent. These subproblems are based on the structure of the particular SSAT instance being solved; the variable ordering in the prefix and the relationships among the variables induced by the clauses dictate the composition of these building blocks for the final solution.

DC-SSAT solves each of these subproblems using a DPLL-based algorithm to generate a set of *viable partial assignments* (VPAs) that satisfy the clauses in that subproblem, and then uses these VPAs to construct the solution to Φ . Note that DC-SSAT does not save redundant VPAs and each VPA saved is necessary to construct the solution. This is in contrast to ZANDER, which saves every solved subproblem, but often uses only a small percentage of cached subproblems. The VPAs saved by DC-SSAT almost always require 1-2 orders of magnitude less space than the subproblems saved by ZANDER. Roughly speaking, in the same way that ZANDER does a depth-first search on variable assignments, DC-SSAT does a depth-first search on VPAs. Intuitively, one can think of the solution tree for Φ as the best combination of VPAs that are compatible, as defined in [MB05]. A further optimization uses the VPAs generated by earlier subproblems to constrain the VPAs generated by the remaining subproblems.

DC-SSAT outperforms ZANDER by several orders of magnitude with respect to both time and space. The results of a comparison of DC-SSAT and ZANDER on a set of COPP-SSAT problems, adapted from [ML03, HSAHB99], are reported in [MB05]. Although DC-SSAT could be generalized to solve SSAT encodings of partially observable planning problems, the linear structure that DC-SSAT takes advantage of is not as pronounced in such problems—there are connections between non-adjacent blocks of variables—and this would likely reduce the effectiveness of this approach.

The DC-SSAT approach is similar to a number of other approaches. First, DC-SSAT is similar to SAT cut-set approaches that attempt to divide the SAT problem into subproblems by instantiating a subset of variables. In a COPP-SSAT problem, the variables shared by subproblems can be viewed as a cut-set that decomposes the problem. Instead of instantiating variables in the cut-set and propagating this information, however, DC-SSAT uses these variables as a conduit for information between adjacent subproblems. These variables can also be viewed as constraints for combining VPAs into a solution. This is due to the fact that, unlike a SAT problem, where the goal is to find satisfying assignments for the subproblems that can be linked through a common cut-set instantiation, DC-SSAT must find all satisfying assignments for each subproblem and determine the optimal combination of these. Because of the ordering on the variables and the structure of any possible solution, DC-SSAT can order the subproblems and use the solutions of earlier subproblems to restrict the search for solutions in subsequent subproblems. This interplay between subproblems, mediated by a subset of common variables, is similar to the *delayed cut variable binding* technique of [PV96] that tries to bind as few of the cut-set variables as possible while still satisfying the subproblems, searching among alternate assignments of the cut-set variables (starting with alternate assignments of the already instantiated cut-set variables) if this proves impossible.

Second, DC-SSAT is similar to the *bucket elimination* algorithm for solving SAT problems described in [Dec99]. In this approach, the clauses of the SAT instance are partitioned into n buckets (one for each variable) according to an arbitrary variable ordering v_1, v_2, \dots, v_n such that bucket i contains all the clauses whose highest literal is v_i . The algorithm then performs resolution on each bucket from bucket n down to bucket 1, putting resolvents in the appropriate bucket according to the variable ordering. When this *directional resolution* has been completed, a satisfying assignment, if one exists, can be generated without backtracking.

Finally, DC-SSAT is similar to the *component caching* approach to solving #SAT problems described in [SBB⁺04]. Unlike the component-caching algorithm, however, in which the solutions to disjoint problem components are computed and saved dynamically and used to calculate the solution to the original #SAT problem, DC-SSAT builds the solution to the COPP-SSAT problem using the solutions to components that share relatively few variables and that are created at the outset.

34.6.4. Approximation Algorithms

Approximation algorithms are often used to avoid the computation of exact answers to difficult optimization problems. MAJSAT instances are very naturally approximated using stochastic sampling (Section 34.6.4.1) and this sampling technique can be extended to provide an approximation algorithm for general SSAT instances (Section 34.6.6). Unfortunately, this approach does not reduce the complexity of solving the general SSAT problem defined in Section 34.2. Even approximating $\Pr^*[\Phi]$ for an instance of the general SSAT problem is hard. Condon et al. [CFLS97] showed that, given an SSAT instance Φ , there exists a constant $c > 0$ such that approximating $\Pr^*[\Phi]$ to within ratio 2^{-n^c} is PSPACE-hard (where n is the number of Boolean variables in Φ). However, SSAT encodings of finite-horizon probabilistic planning problems (described in Section 34.4.1) have relatively few quantifier alternations and the number of alternations is independent of the number of variables in the SSAT encoding, depending instead on the horizon (number of steps) in the encoded planning problem. It is not known whether the SSAT problem, restricted in this way, is also hard to approximate; APPSSAT, a technique for approximating the solution to such a problem is described in Section 34.6.5.

34.6.4.1. An Approximation Algorithm for Random MAJSAT Problems

The answer to a MAJSAT instance Φ can be approximated by generating some number of assignments to the random variables according to their probabilities and then estimating $\Pr^*[\Phi]$ from that sample. The following lemma directly gives an approximation algorithm for MAJSAT [LMP01]. Choose a set W of w assignments to the randomized variables proportional to their probability. The value of the policy is estimated as the sum of the probabilities of the satisfying sampled assignments divided by the sum of the probabilities of all the sampled assignments. Let v be the true value of $\Pr^*[\Phi]$ and \hat{v} be the estimate found via sampling. A direct application of Chernoff bounds shows that $w = O(\frac{1}{\epsilon^2} \log(\frac{1}{\delta}))$ samples are sufficient to produce an estimate no further than ϵ away from the true value with probability $1 - \delta$:

Lemma 3. Let $\epsilon > 0$ be a target approximation error. The probability that $|v - \hat{v}| > \epsilon$ is less than $2e^{-2\epsilon^2 w}$.

34.6.4.2. An Approximation Algorithm for Random SSAT Problems

As described in Section 34.6.4.1, approximating $\Pr^*[\Phi]$ for a MAJSAT is straightforward. The existence of existential variables in an SSAT instance makes it impossible to apply this sampling technique directly, but sampling can still be used to create an approximation algorithm. The `sampleevalssat` algorithm, presented in [LMP01], uses random sampling to select a subset of randomized variable instantiations. This subset of randomized variable instantiations limits the size of a possible solution by selecting a subset of paths to explore in the tree of assignments. In effect, this converts the SSAT problem to a (possibly exponentially larger) SAT problem. The `sampleevalssat` algorithm then systematically searches

```

sampleevalssat( $\Phi, W$ ) := {
  if  $\phi$  contains an empty clause:
    return 0;
  if  $\phi$  is the empty set of clauses:
    return 1;

  if  $Q_i = \exists$  {
    return max(sampleevalssat( $\Phi(\overline{x}_i), W$ ),
              sampleevalssat( $\Phi(x_i), W$ ))
  }
  else if  $Q_i = \forall$ , {
    Let  $W_0$  be the samples in  $W$  that assign  $x_i$  false
    Let  $W_1$  be the samples in  $W$  that assign  $x_i$  true
    return  $\frac{|W_0|}{|W|}$  sampleevalssat( $\Phi(\overline{x}_i), W_0$ ) +
           $\frac{|W_1|}{|W|}$  sampleevalssat( $\Phi(x_i), W_1$ )
  }
}

```

Figure 34.2. A stochastic sampling algorithm for SSAT problems.

for the existential variable assignments that maximize the probability of satisfaction of the partial tree formed by these samples. Given a sample W of size w of assignments to the randomized variables in an SSAT formula, the **sampleevalssat** algorithm (pseudocode from [LMP01] provided in Figure 34.2) computes the maximum possible $\text{Pr}^*[\Phi]$ given the sample W .

The running time of **sampleevalssat** is $w2^E$, where E is the number of existential variables in the formula, and [LMP01] also show that:

$$w = O\left(\frac{1}{\epsilon^2} \left(\log\left(\frac{P}{\delta}\right)\right)\right)$$

samples are sufficient to be sure with probability $1 - \delta$ of having an estimate no further than ϵ away from the true value, where P is the number of possible assignment trees for the SSAT instance. The number of samples needed to obtain an accurate estimate of the value of the formula $\text{Pr}^*[\Phi]$ with high probability is polynomial for SAT, MAJSAT, and E-MAJSAT, but exponential for ASSAT [LMP01].

Tests of **sampleevalssat** in [LMP01] on a set of problems drawn from $\mathcal{F}_m^{k,n}$ (the distribution of formulas obtained by selecting independently m clauses of size k over n variables), restricted to problems in which all n variables appear in the formula, over a range of number of quantifier alternations and a range of sample sizes show that the mean and variance of the squared error approach zero as the number of samples increases, approaching the set of all possible assignments to the randomized variables.

Many other researchers have explored approximations for probability-based problems. [KMN02] show how approximately optimal actions can be chosen with high probability in infinite-horizon discounted Markov decision processes. They show that it is sufficient to consider the first H actions in the sequence, for an appropriate choice of H . The space of all H -step action sequences is then evaluated

using random sampling. The `sampleevalssat` algorithm was directly inspired by this work [LMP01]. [Rot96] studied the complexity of inference in belief networks and showed that computing the probability of a query node (similar to evaluating a MAJSAT formula) is $\#P$ -complete. In addition, approximating the probability of a query node to within a multiplicative factor of its true value is just as hard as computing the exact value. In contrast, Lemma 3 in Section 34.6.4.1 shows that the complexity of getting within an *additive* factor for MAJSAT is polynomial.

34.6.5. Approximation Algorithms for Structured Problems

APPSSAT is an approximation technique for solving SSAT encodings of finite-horizon, goal-oriented, probabilistic propositional contingent planning problems [Maj07]. APPSSAT is an anytime planner based on ZANDER (see Sections 34.4.1 and 34.6.3.1). Unlike ZANDER, which, in effect, looks at randomized variable instantiations at a particular time step based on the instantiation of variables (particularly existential variables that encode actions) at previous times steps (i.e. earlier in the quantifier ordering), APPSSAT considers the most likely instantiations of the randomized variables that encode the uncertainty in the planning problem and incrementally constructs the optimal solution tree by updating the probabilities of the possible assignments paths in that tree as it processes the instantiations of the randomized variables. In a single iteration, APPSSAT: 1) selects the randomized variable instantiation with the next highest probability p , 2) uses a SAT solver to find all the satisfying assignments that extend this randomized variable assignment, 3) installs each such satisfying assignment into the solution tree and updates, as necessary, the probabilities of success of the actions and the optimal action at each point in the tree, and 4) uses a kind of threshold pruning to compute the probability of success of the optimal plan so far and compares it to the target threshold probability. Given more time, less likely instantiations are considered and the solution is revised as necessary.

APPSSAT, by enumerating complete instantiations of the randomized variables that encode domain uncertainty in descending order of probability, examines, in effect, the most likely outcomes of all actions at all time steps. Instantiating all the randomized variables also has the benefit of reducing the SSAT problem to a number of much simpler SAT problems that can be solved efficiently using a state-of-the-art SAT solver like zChaff [MMZ⁺01]. An alternate approach noted in [Maj07], but rejected due to considerations of excessive overhead, is to instantiate randomized variables based on the instantiation of variables (particularly action variables) at previous times steps.

In some cases, a solution constructed to address a relatively low percentage of possible instantiations will succeed for instantiations not explicitly considered as well, and may return an optimal or near-optimal solution. Experimental results indicated that this approach can sometimes find sub-optimal plans in cases in which ZANDER is unable to find the optimal (or any) plan. In a domain in which all the random variable instantiations are approximately equally likely, however, this approach will perform poorly. And, given APPSSAT's performance on the small test problems in [Maj07], it seems unlikely that this approach could scale up to large, real-world problems without significant modification.

34.6.6. Non-Systematic Algorithms for Random Problems

Non-Systematic search techniques, such as stochastic local search (SLS), consider a subset of potential solutions. SLS searches for a solution by starting with a candidate solution and iteratively exploring the solution space. In one iteration, SLS uses local information to stochastically select another candidate solution that is “close to” the current candidate (according to some metric) and moves to that candidate solution [HS05]. While not complete, SLS algorithms can often solve significantly larger instances of a problem than a systematic algorithm. SLS algorithms for SAT have been extensively investigated since the early success of such algorithms [SLM92, SKC96, MSK97]. (See Part 1, Chapter 6 for a discussion of local search techniques for satisfiability.) SLS techniques have also been incorporated in algorithms for QBF [GHR03, HR05].

Using SLS for SSAT problems is complicated by the fact that an SSAT solver needs to systematically consider all possible assignments to solve the SSAT problem exactly. The stochastic sampling algorithm, `sampleevalssat` (Section 34.6.4.2), however, has been used to create `randevalssat`, an algorithm that is both an approximation algorithm and a non-systematic algorithm for solving SSAT instances [LMP01]. Like the `sampleevalssat` algorithm, the `randevalssat` algorithm limits the size of a possible solution tree—and converts the problem to a lower complexity class problem (SAT)—by randomly sampling to select a subset of randomized variable instantiations. But, instead of systematically searching for the existential variable assignments that maximize the probability of satisfaction of the tree formed by these samples, `randevalssat` uses SLS to look for these existential variable assignments.

This is done by converting the sampled randomized variable instantiations into a two-level Boolean formula called a *treeSAT* formula. A satisfying assignment to the *treeSAT* formula corresponds to a setting of the decision variables that satisfies the original SSAT formula for every random variable instantiation sampled. In general, it will not be possible to find such a setting, so [LMP01] use stochastic local search. This is similar to the scenario-based approach to stochastic constraint programming of [TMW06], briefly described in Section 34.7. Tests on sets of problems drawn from $\mathcal{F}_m^{k,n}$ showed the mean and variance of the squared error decreasing as the number of samples increased, in some cases approaching the accuracy of `sampleevalssat` (Section 34.6.4.2) in less than 25% of the time.

There are a number of problems, however, with the `randevalssat` algorithm. First, like all SLS algorithms, `randevalssat` can get stuck in local optima. Second, the memory requirements of the algorithm to store the *treeSAT* formula can be prohibitively large. In fact, there are SSAT problems for which the algorithm needs provably large samples (see [LMP01] for an example of such a problem). Third, `randevalssat` does not return an answer whose correctness can be readily checked. Fourth, the algorithm returns a partial solution that specifies existential variable settings only for those randomized variable instantiations that were sampled. Finally, `randevalssat` is unsuited to solving SSAT encodings of planning problems (Section 34.6.3.1). The `randevalssat` algorithm would require the observations to be represented by randomized variables, so a random instantiation of the randomized variables describes an observation sequence as well as

an instantiation of the uncertainty in the domain. The observation sequence described, however, may not be observationally consistent, and these inconsistencies can make it impossible to find a solution, even if one exists. The APPSSAT algorithm (Section 34.6.5) suggests one way to deal with this problem.

Some of these issues could be addressed by using the probabilities of the random variables to direct the construction of the partial solution tree. A more substantial modification would be to iteratively build the partial solution tree by using the solution of the partial tree in a given iteration to construct a better partial solution tree (and solution) in the next iteration.

34.7. Stochastic Constraint Programming

We have cited related work in relevant sections throughout this chapter. Here we discuss *stochastic constraint programming* (SCP), a technique developed to model and solve combinatorial decision problems involving uncertainty [Wal02, TMW06]. SCP, a combination of traditional constraint satisfaction, stochastic integer programming, and stochastic satisfiability, generalizes the probabilistic planning as stochastic satisfiability paradigm by allowing variables with arbitrary domains and arbitrary constraints over those variables. A *stochastic constraint satisfaction program* (SCSP) is a decision form of stochastic constraint programming. As defined in [TMW06], an SCSP is a 6-tuple $\langle V, S, D, P, C, \theta \rangle$ where:

1. V is a set of *decision variables*,
2. S is a set of *stochastic variables*,
3. D is a function mapping each variable in V and S to a domain of possible values,
4. P is a function mapping each variables in S to a probability distribution for its domain,
5. C is set of *constraints* over V defined in the standard CSP manner,
6. $H \subseteq C$ is the set of *chance constraints*, i.e. constraints containing at least one stochastic variable, and
7. for each chance constraint $h \in H$, $0 \leq \theta_h \leq 1$ is a *threshold probability* specifying the probability with which that constraint must be satisfied.

In an m -stage SCSP, V and S are partitioned into disjoint subsets V_1, V_2, \dots, V_m and S_1, S_2, \dots, S_m . Each stage contains a set of decision variables and a set of stochastic variables, modeling the interaction between a set of decisions and an uncertain environment. In stage i , the decision variables in V_i are those variables whose values can be set by a solver (e.g. production levels) and the stochastic variables in S_i , whose values are set according to their associated probability distributions, encode the uncertainty in the environment that affects the results of those decisions (e.g. demand for the products). Decision and stochastic variables are clearly analogous to the existential and randomized variables in an SSAT problem, and the partition and ordering of the subsets of V and S are analogous to a $2m$ -block SSAT problem prefix of $V_1 S_1 V_2 S_2 \dots V_m S_m$, and the solution of the m -stage SCSP is similar to the solution of the $2m$ -block SSAT problem. Informally, a solution specifies settings for the decision variables in V_i for each setting of the stochastic variables in $\cup_{j=1}^{i-1} (V_j \cup S_j)$ such that the hard constraints

$C - H$ are satisfied and each chance constraints $h \in H$ is satisfied with probability at least θ_h .

SAT and CSP, the deterministic versions of SSAT and SCP, are both NP-complete and the correspondences between SSAT and SCP problems and their respective complexities are close and immediate. A 1-stage SCSP in which all the variables are stochastic is the SCP version of MAJSAT, and both are PP-complete. A 1-stage SCSP in which a set of decision variables is followed by a set of stochastic variables is the SCP version of E-MAJSAT, and both are NP^{PP}-complete. Finally, an m -stage SCSP is the SCP version of SSAT, and both are PSPACE-complete.

SCSPs are expressed in the *stochastic optimization programming language* (stochastic OPL) [TMW06], an extension of OPL, the constraint modeling language of [VMR99]. Stochastic OPL supports stochastic variables and constraints as well as their deterministic counterparts. Stochastic OPL supports both deterministic and stochastic optimization (e.g. optimize the expectation of an objective function) and risk modeling (e.g. maximize the absolute deviation function, minimize the difference between the best- and worst- case objective function values, maximize the minimum value of the objective function).

The semantics for a stochastic constraint program can be either *policy-based* [Wal02] or *scenario-based* [TMW06]. The policy view is based on a *policy tree*, a tree representing a policy that specifies values for the decision variables (similar to an SSAT solution tree). Each decision variable node has one child (the value chosen by the policy) and each stochastic variable node has a child for each possible value in that variable's domain. The probability of satisfaction of the policy is defined as the sum of the leaf values weighted by their probabilities. A leaf is assigned the value 1 (0) if the partial assignment specified by the path from the root to that leaf satisfies (does not satisfy) all the constraints. The probability of a leaf is the probability that the stochastic variables on that path have the values specified by the partial assignment. Note that the probability of satisfaction of an SSAT solution tree could be defined in the same way. Satisfying policies—in particular, the policy that has the maximum probability of satisfaction—can be found using backtracking and forward checking algorithms. The algorithm described in [Wal02] that is associated with this policy-based view is essentially the *evalssat* algorithm described in Section 34.6.2.1 with forward checking.

Each path in a policy tree can also be thought of as a scenario. The scenario view makes this explicit by constructing a *scenario tree* whose paths correspond exactly to all possible assignments of the discrete stochastic variables in the SCSP. For each path, instantiating those variables to the values specified by that path yields a standard, deterministic CSP. When this is done for each path in the scenario tree, a collection of deterministic CSPs is produced, each of which has V as its set of variables. The values for some of these variables must have the same value in any solution to the SCSP, e.g. the decision variables in V_1 must have the same value in any scenario. With the addition of appropriate constraints ensuring the necessary agreement among shared variables, however, the collection of CSPs can be solved using standard CSP solution techniques, yielding a solution to the underlying SCSP. In spite of the fact that the number of scenarios and, thus, the

collection of CSPs that need to be solved, grows exponentially with the number of stages in the SCSP, [TMW06] report better performance using the scenario-based approach on a production problem from [Wal02] than a policy-based approach using either backtracking or forward checking.

34.8. Future Directions

SSAT research is still in its early stages; there are numerous opportunities for further work, some of which we describe here.

34.8.1. Data Structures

More sophisticated data structures in which to store the matrix of the SSAT instance would almost certainly improve the efficiency of an SSAT engine. *Lazy data structures*, in particular *watched literals*, have been used to great advantage in SAT engines [MMZ⁺01]. Three different types of watched data structures for QBF were investigated in [GGN⁺03]: literals, quantifiers, and clauses. In tests on structured and randomly generated problems, watched literals provided modest performance speed-up (less than an order of magnitude) that was positively correlated with clause size, watched quantifiers provided no improvement, and watched clauses provided the most performance speed-up (as much as two orders of magnitude).

The *trie* data structure has been used to represent SAT problems, and several advantages have been claimed for this approach [ZS00], including automatic elimination of duplicate clauses when the trie is constructed, reduced memory requirements, more efficient unit propagation. These advantages would be obtained in the SSAT setting as well.

34.8.2. Branching Heuristics

As noted in Section 34.6.2.1, branching heuristics for SSAT problems are limited by the requirement that, except for instances of unit propagation and pure variable elimination, variables in the left-most block of the prefix must be assigned values before variables that appear later in the prefix. Branching heuristics come into play, however, in deciding which variable in the left-most block will be branched on next. The DPLL-XSSAT algorithm described in Section 34.6.2.1 merely selects the first variable that appears in the SSAT problem's current prefix; in a randomly generated problem, this is equivalent to following a static, random ordering. SSAT algorithms that solve encodings of probabilistic planning problems, such as MAXPLAN, ZANDER (described in Section 34.6.3.1), and DC-SSAT (described in Section 34.6.3.2), use a branching heuristic that gives priority to variables with lower time indices, but do not specify an ordering for variables within blocks of similarly quantified variables that have the same time index, again using a static, random ordering. Using a random ordering is likely to be insignificant in small problems, but in problems with large blocks, such as planning problem encodings of large real-world planning problems, a branching heuristic that uses a more sophisticated branching heuristic could provide a significant performance gain.

Experiments reported in [LMP01] and briefly described in Section 34.6.2.1 provide evidence that performance gains can be obtained using straightforward adaptations of several common branching heuristics for SAT problems. These heuristics exploit characteristics such as how many clauses a variable appears in, how small these clauses are, and the ratio of positive to negative instances of the variable. It might be possible, however, to develop more effective branching heuristics that take into account other features of SSAT problems, such as the ordering of variables in the prefix or the probabilities associated with the randomized variables.

34.8.3. Graph Representations

It might be useful to study SSAT problems by looking at their hypergraph representations. In one possible hypergraph representation, the vertices represent clauses and a hyperedge represents a variable shared by the subset of clauses the hyperedge connects. There are a number of well-studied characteristics of hypergraphs that appear to have a relationship to the difficulty of SSAT problems [New03]. This approach could yield insights into the nature of SSAT problems and provide the basis for new, more efficient SSAT solvers. For example, DC-SSAT, the solver described in Section 34.6.3.2, is able to solve COPP-SSAT problems so efficiently because, when viewed as a hypergraph, these problems exhibit a high degree of *community structure*, i.e. the existence of groups of nodes that have a high density of connections within the groups but relatively few connections between groups [New03]. In fact, a decomposition of a hypergraph representation of an SSAT encoding of a COPP-SSAT problem that minimizes connections between the components of the decomposition yields exactly the problem decomposition used by the DC-SSAT algorithm. Is it possible to characterize when the DC-SSAT approach will be beneficial in the case of general SSAT problems? Variables that are far apart in the prefix, but share a clause, may induce the solver to explore a long, unproductive path. Assignment compatibility issues that were local in a COPP-SSAT problem become global in the general case. [Wal01] investigated the impact of the connection topologies of graphs associated with real-world search problems and related this to the notion of a *backbone* (variables that have the same value in all solutions). What role do these notions play in SSAT problems? What kinds of structure exist in SSAT encodings of real-world problems and how can that structure be exploited?

34.8.4. Nonchronological Backtracking

Results in [Maj04] indicate that the benefits of nonchronological backtracking (NCB) are not as easy to realize in SSAT as QBF. One response to that problem—monitor the solution process and turn off NCB when it appears that the overhead of NCB will outweigh its benefit—is described in [Maj04]. A more efficient way of creating and manipulating reasons, e.g. using bit vectors, would reduce the overhead of NCB. Since a SAT reason can be any subset of the randomized literals necessary for the satisfaction of some clause, choosing that subset with the aim of minimizing its size may not be the best strategy. [Maj04] suggests

that, where there are choices, a criterion that favors randomized literals that appear in clauses with fewer existential literals may improve the efficiency of NCB, since such literals will have less of a tendency to pull existential literals into the reason during backtracking. The poor performance of NCB on SSAT encodings of planning problems, in which most clauses have at most one randomized literal, lends support to this notion. It may also be possible to develop branching heuristics that reduce the average size of the reasons created, thereby improving the effectiveness of NCB.

34.8.5. Component Caching

Work in [BDP03b, SBB⁺04], briefly described in Section 34.6.2.1, suggests that component caching would be much more effective than the formula caching currently used in DPLL-XSSAT. The superior performance of DC-SSAT, which uses an approach similar to component caching to solve SSAT encodings of completely observable probabilistic planning problems, tends to confirm this and suggests that it would be useful to develop more refined component caching approaches.

34.8.6. Learning

Learning contributes significantly to the speed of state-of-the-art SAT solvers (see Part 1, Chapter 4 for more on this) and it seems likely that learning could improve the performance of SSAT solvers as well. For example, clause learning is an important component of nonchronological backtracking (NCB) in SAT problems. Recording reasons as they are created in order to prevent the solver from rediscovering them in other branches of the computation has been used to significantly enhance the performance benefits obtained from reason creation in both SAT [BS97] and QBF [GNT02] solvers. This technique has the potential to improve the effectiveness of NCB in an SSAT solver (Section 34.6.2.1). UNSAT reasons can be recorded as additional clauses that prevent partial assignments leading to unsatisfiability; recording SAT reasons is not as straightforward [Maj04].

[SM07] use machine learning techniques in a search-based QBF solver to predict and select the best variable selection heuristic to use from a portfolio of heuristics, given the current subproblem. This approach produced positive results both in terms of run times and percentage of problems solved on a suite of benchmarks, and it is likely that this approach would be effective for SSAT problems as well and could be adapted for SSAT without significant alterations.

34.8.7. Stochastic Local Search

[Maj00] describes several ways that stochastic local search could be incorporated into an SSAT solver other than the approach described in Section 34.6.6. The *randevalssat* algorithm could be used as a subroutine in an algorithm that periodically restarts with a new set of samples. Stochastic local search could be used repeatedly to find many satisfying assignments that are used to create a solution tree; this is the approach taken by WALKSSAT, an algorithm described in [ZM05]. WALKSSAT generates an initial path in a potential solution tree by

treating all the variables in the SSAT problem as existential variables and using WALKSAT to search for a solution. WALKSSAT then uses WALKSAT as a subroutine to expand nodes on this path and on subsequently created paths. Node expansion continues until all nodes are expanded, a specified minimum probability of satisfaction is met, or a specified maximum number of expansions is reached. Tests of WALKSSAT were inconclusive. The algorithm could almost certainly be improved by using a better heuristic to direct the order of node expansions; currently WALKSSAT expands the most recently generated node first, like a depth-first search. This also weakens the idea of WALKSSAT as a local search algorithm.

A more promising approach, currently being developed by the author, combines local search and systematic search, using WALKSAT to identify possible paths in a solution tree and depth-limited DPLL-style systematic search to expand the tree. Stochastic local search has also been used as a component in solvers for 2-QBF [HR05] and QBF [GHR03]. In both cases, local search is used to quickly find satisfying assignments to SAT problems that result either from instantiating the universal variables or from treating the universal variables as existential variables, and these approaches could possibly be adapted for SSAT.

34.8.8. Approximation Techniques

A possible approximation technique for SSAT would convert the SSAT instance into a SAT instance by treating the randomized variables as existential variables with values determined by the rounded probabilities associated with those variables, solving the resulting SAT problem efficiently, and then gradually reintroducing uncertainty to improve the quality of the solution. It is not clear, however, how to reintroduce the uncertainty without sacrificing the efficiency gained by removing it.

34.8.9. Algebraic Approaches

Algebraic approaches have been applied to SAT. [CEI96] showed that the Gröbner basis algorithm could be used as the basis for a heuristic for unsatisfiability testing in SAT problems. More recently, [CK07] described a method that uses Gröbner bases to preprocess the SAT CNF formula, adding new constraint information. Preliminary results indicated that, while the preprocessing time often outweighs the reduction in solution time, there appears to be some potential in using the approach on structured problems. It might be useful to investigate this type of approach in the SSAT setting.

34.8.10. Message Passing Algorithms

Survey propagation is an algorithm for solving SAT instances that has shown promise for solving the hardest random k -SAT instances, which, empirically, are concentrated in a sharply defined region corresponding to a clause/variable ratio that varies with the value of k . Like the belief propagation algorithm for belief network inference [Pea88], survey propagation is a message-passing algorithm, but, unlike belief propagation, in which the messages between cliques of variable

nodes in the belief network contain conditional probability information, messages in survey propagation are passed between nodes representing variables and clauses in a *factor graph* representation of the SAT instance, and contain information about the expected values of variables in a solution [BMZ05]. These messages are *surveys*, or aggregations, of simpler *warning messages* that, very roughly speaking, “warn” a clause c containing literal l when there are a number of other clauses containing \bar{l} and, as a result, would like the value of $|l|$ to be set such that l would be **false**, making it more likely that c would become empty. This probabilistic information about the solution is used to fix the values of one or more variables, thus simplifying the problem. This process is iterated until the problem is sufficiently simplified so that it can be solved using a standard SAT solver.

Survey propagation and DC-SSAT (see Section 34.6.3.2) are related. (There are also similarities between survey propagation and bucket elimination, since bucket elimination is similar to a directional version of belief propagation [Dec99]. In fact, all of these algorithms—survey propagation, belief propagation, bucket elimination, and DC-SSAT—operate by distributing locally calculated information throughout the problem and using that information to guide the search for a solution.) Like the messages in survey propagation, the VPAs (viable partial assignments) that are calculated by DC-SSAT and passed between the COPP-SSAT subproblems contain information about how these smaller pieces of the problem restrict the values of variables in any possible solution. A direct generalization of DC-SSAT to problems without the strict linear structure of a COPP-SSAT problem, is unlikely to be as effective as DC-SSAT on SSAT encodings of COPP-SSAT problems. It is possible that the survey propagation technique could provide the basis for a DC-SSAT-like algorithm that could solve general SSAT problems as effectively as DC-SSAT solves SSAT encodings of COPP-SSAT problems.

34.8.11. Adapting Other QBF Techniques

The close relationship between stochastic satisfiability and quantified Boolean formulas (see Section 34.2.1) suggests that some of the techniques that have been developed to solve QBF problems in addition to those mentioned in Sections 34.8.4, 34.8.6, and 34.8.7 could profitably be adapted to solve SSAT problems.

A number of researchers have shown that inference can be an effective tool to solve QBF problems. [KBKF95] introduced *Q-resolution*, a procedure that can eliminate multiple universal variables. sKizzo [Ben04] is a QBF solver that combines variable elimination and search. [CH05] developed Q-PREZ, a QBF solver that uses resolution, mitigating the potential exponential clause growth by partitioning the clauses and storing them in zero-suppressed binary decision diagrams. The QBF solver quantor [Bie05] applies Q-resolution and universal variable expansion until all universal variables are eliminated and the resulting equisatisfiable formula can be solved by a SAT solver. [GNT06] introduced Q-resolution on terms for formulas in disjunctive normal form and related this type of resolution and the original Q-resolution on clauses [KBKF95] to DPLL-based approaches. [SB06] showed how extended binary clause reasoning (hyper-binary

resolution) can be adapted for QBF problems in the solver 2clsQ. (See Part 2, Chapter 30 for a discussion of resolution and variable elimination in QBF.)

Aside from unit propagation and pure variable elimination, inference techniques are absent from current SSAT solvers. The similarity of SSAT and QBF suggests that the inference techniques employed by various QBF solvers could be adapted for use in SSAT solvers. Adapting these techniques, however, is complicated by the fact that the randomized variables in an SSAT problem impose less stringent demands than do universal variables and, consequently, weaken, or prohibit the use of some of the inference techniques employed in QBF solvers. For example, [Bie05] uses Q-resolution and *universal variable expansion* to eliminate variables. Q-resolution is standard resolution followed by *forall reduction*, in which, roughly speaking, universal variables can be removed from a clause if the clause contains no existential variables that come after any of the universal variables in the prefix. The clause must be satisfied by one of its existential literals, since, if all of them are **false**, a contradictory clause of universal literals remains. In an SSAT problem, the standard resolution would be possible, but randomized variables cannot be eliminated in the same way that forall reduction eliminates universal variables. The elimination of randomized variables in a clause, even if there are no existential variables in the clause that appear after any of the randomized variables in the prefix, could erroneously reduce the overall probability of satisfaction, since, even if all its existential literals are **false**, the clause may still be satisfiable with some probability greater than zero. This is a significant problem since this kind of reduction is an important way to control the exponential growth in formula size that variable elimination can produce.

Universal variable expansion is also not possible. In [Bie05], universal variable expansion is applied to a formula in which the last two blocks in the prefix are a universal block followed by an existential block. Roughly speaking, the elimination of a universal variable through expansion is accomplished by duplicating the clauses containing the universal and existential variables, replacing the existential variables in the second set with new variables, and then, in one set of clauses, setting the universal variable to be eliminated to **false** and propagating the effect, and, in the other set, setting the universal variable to **true** and propagating the effect. This can reduce the number of clauses in the matrix as well as eliminating the universal variable. Unfortunately, due to the probabilistic nature of randomized variables, eliminating them in this way can lead to an incorrect reduction in the probability of satisfaction.

Essentially, randomized variables are harder to eliminate than universal variables. In order to eliminate randomized variables, it might be possible to tag the clause groups from which random literals have been removed with information that a SAT solver could use when solving the eventual SAT problem, to produce the correct solution to the SSAT problem. If each group of clauses were tagged with 1) the values of the randomized variables that eliminated the randomized literals in that group of clauses, and 2) the product of the probabilities of those randomized variable settings, the SAT solver could use this information to calculate the probability of satisfaction (the probability of satisfaction would need to be reduced when one of the clauses in a group becomes **false**) and to construct the solution tree. Adapting variable elimination techniques for belief networks

[Pea88, Jor99] might be possible here.

QBF and SSAT share the restriction imposed by the variable ordering in the prefix. Although the variable ordering within blocks of similarly quantified variables can be ignored, the necessity to respect the ordering of the blocks in the prefix is a major factor contributing to the difficulty of solving SSAT problems. [AS04] describe a way, however, to remove this restriction in QBF: as satisfying assignments are found using a DPLL-based SAT solver (treating all variables as existential variables), these assignments are combined in a reduced ordered binary decision diagram (ROBDD) in which the ordering of variables is the same as the variable ordering in the prefix of the QBF instance. If the set of assignments stored in the ROBDD is a solution, then the ROBDD reduces to the 1-terminal node. Such an approach could possibly be adapted to SSAT instances by using some form of algebraic decision diagrams, instead of ROBDDs, to allow probabilities of satisfaction to be represented and manipulated. The optimization nature of an SSAT problem, however, makes the stopping criterion unclear.

34.8.12. Combining Logic-Based Methods and Optimization Methods

Researchers from both the artificial intelligence and operations research communities have begun to develop hybrid representations and solution techniques that integrate logic-based modeling and techniques with mathematical modeling. [HOTK00] has proposed a framework for unifying constraint satisfaction and optimization methods based on the similarity of the search/inference duality in constraint satisfaction and the strengthening/relaxation duality in optimization. Using branching algorithms as an example, [HOTK00] points out that branching on a variable can be viewed as search, from a CSP perspective, or as a strengthening of the original problem by restricting the value of a variable, from the optimization perspective. Furthermore, the reduction of variable domains in constraint satisfaction obtained through constraint propagation (inference) after a variable has been assigned a value is analogous to a relaxation of the problem at that node in the tree in an optimization approach. The use of logic-based methods for optimization and optimization methods for logical inference has been elaborated in [Hoo00] and [CH99], respectively.

As described in Section 34.7, [TMW06, Wal02] have integrated ideas from constraint programming and stochastic satisfiability with ideas from stochastic integer programming to produce stochastic constraint programming (SCP). No work has been done to compare the relative strengths and weaknesses of the SCP-based approach and the SSAT-based approach to decision making under uncertainty. Unlike the SSAT approach, the SCP approach can express global and arithmetic constraints; it can also express variables with non-binary domains more easily. On the other hand, it is likely that the SSAT-based approach can take advantage of some of the same techniques that have produced extremely fast SAT solvers in recent years. In the deterministic setting, a study of mappings between CSPs and SAT problems [Wal00] found that the efficiency of the standard DPLL algorithm for SAT instances and the forward-checking (FC) and maintaining-arc-consistency (MAC) algorithms for CSPs, as measured by the size of the search tree generated, was highly dependent on the encoding used. On

some encodings, DPLL dominated the other two algorithms, while on others, FC and/or MAC dominated. A similar study comparing SCSPs and SSAT problems would help delineate the advantages and disadvantages of these two approaches.

[DGP04, DGLP04, DGH⁺05] have investigated the benefits of pseudo-Boolean representations of SAT problems extensively. (See Part 2, Chapter 28 for a discussion of pseudo-Boolean constraints and satisfiability.) [ZKC01] have developed LPSAT, an efficient linear-programming-based SAT solver that uses mixed integer linear programming (MILP) to solve a circuit design problem involving both Boolean and arithmetic components. In order to combine these different components into the MILP framework, they represent the Boolean components as an integer linear program, rather than a SAT problem, and the arithmetic components as linear arithmetic constraints. [Wal97] has developed a stochastic local search algorithm for pseudo-Boolean representations of SAT problems. And [WW00, KW99] have developed techniques that incorporate linear programming into SAT-based planning. It would be worthwhile to investigate the possibility of using these and other representations and solution techniques borrowed from operations research in the SSAT setting.

References

- [AS04] G. Audemard and L. Saïs. SAT based BDD solver for quantified boolean formulas. In T. Khoshgoftaar, editor, *Proceedings of the Sixteenth IEEE International Conference on Tools with Artificial Intelligence*, pages 82–89. IEEE Computer Society, 2004.
- [Bal83] B. W. Ballard. The *-minimax search procedure for trees containing chance nodes. *Artificial Intelligence Journal*, 21(3):327–350, 1983.
- [BDP03a] F. Bacchus, S. Dalmao, and T. Pitassi. Algorithms and complexity results for #SAT and Bayesian inference. In *Proceedings of The Forty-Fourth Annual IEEE Symposium on Foundations of Computer Science (FOCS-2003)*, pages 340–351, 2003.
- [BDP03b] F. Bacchus, S. Dalmao, and T. Pitassi. DPLL with caching: A new algorithm for #SAT and Bayesian inference. In *Electronic Colloquium on Computational Complexity (ECCC) 10(003)*, 2003.
- [BDP03c] F. Bacchus, S. Dalmao, and T. Pitassi. Value elimination: Bayesian inference via backtracking search. In *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence (UAI-2003)*, pages 20–28. Morgan Kaufmann, 2003.
- [Ben04] M. Benedetti. sKizzo: a QBF decision procedure based on propositional skolemization and symbolic reasoning. Technical Report TR04-11-03, Istituto per la Ricerca Scientifica e Tecnologica (IRST), 2004.
- [Bie05] A. Biere. Resolve and expand. In H. H. Hoos and D. G. Mitchell, editors, *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing*, volume 3542 of *Lecture Notes in Computer Science*, pages 59–70. Springer, 2005.
- [BIPS06] P. W. Beame, R. Impagliazzo, T. Pitassi, and N. Segerlind. Formula

- caching in DPLL. Technical Report TR06-149, Electronic Colloquium in Computation Complexity, 2006.
- [BKPS98] P. W. Beame, R. M. Karp, T. Pitassi, and M. E. Saks. On the complexity of unsatisfiability of random k -CNF formulas. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, pages 561–571, 1998.
- [BMZ05] A. Braunstein, M. Mezard, and R. Zecchina. Survey propagation: an algorithm for satisfiability. *Random Structures and Algorithms*, 27:201–226, 2005.
- [BP96] P. W. Beame and T. Pitassi. Simplified and improved resolution lower bounds. In *Thirty-Seventh Annual Symposium on Foundations of Computer Science*, pages 274–282. IEEE, 1996.
- [BP00] R. J. Bayardo, Jr. and J. D. Pehoushek. Counting models using connected components. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, pages 157–162. AAAI Press/The MIT Press, 2000.
- [BS97] R. J. Bayardo, Jr. and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 203–208. AAAI Press/The MIT Press, 1997.
- [BSW99] E. Ben-Sasson and A. Wigderson. Short proofs are narrow: Resolution made simple. To appear in STOC 1999, 1999.
- [CA96] J. M. Crawford and L. D. Auton. Experimental results in the crossover point in random 3SAT. *Artificial Intelligence Journal*, 81(1-2):31–57, 1996.
- [CEI96] M. Clegg, J. Edmonds, and R. Impagliazzo. Using the Gröbner basis algorithm to find proofs of unsatisfiability. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, pages 174–183, 1996.
- [CFLS97] A. Condon, J. Feigenbaum, C. Lund, and P. Shor. Random debators and the hardness of approximating stochastic functions. *SIAM Journal on Computing*, 26(2):369–400, 1997.
- [CGS97] M. Cadoli, A. Giovanardi, and M. Schaerf. Experimental analysis of the computational cost of evaluating quantified Boolean formulae. In *Fifth Congress of the Italian Association for Artificial Intelligence (AI*IA '97)*, volume 1321, pages 207–218. Lecture Notes in Artificial Intelligence, Springer-Verlag, 1997.
- [CGS98] M. Cadoli, A. Giovanardi, and M. Schaerf. An algorithm to evaluate quantified Boolean formulae. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 262–267. The AAAI Press/The MIT Press, 1998.
- [CH99] V. Chandru and J. N. Hooker. *Optimization Methods for Logical Inference*. John Wiley & Sons, Inc., New York, NY, 1999.
- [CH05] K. Chandrasekar and M. S. Hsiao. Q-prez: Qbf evaluation using partition, resolution and elimination with zbdd. In *Proceedings of the Eighteenth International Conference on VLSI Design*, pages 189–194. IEEE Computer Society, 2005.

- [CK07] C. Condrat and P. Kalla. A Gröbner basis approach to CNF-formulae preprocessing. In *Proceedings of the Thirteenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 618–631. Springer, 2007. Held as part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007.
- [Dec96] R. Dechter. Bucket elimination: A unifying framework for probabilistic inference. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence (UAI-1996)*, pages 211–219. Morgan Kaufmann, 1996.
- [Dec99] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence Journal*, 113:41–85, 1999.
- [DGH⁺05] H. E. Dixon, M. L. Ginsberg, D. Hofer, E. M. Luks, and A. J. Parkes. Generalizing Boolean satisfiability III: Implementation. *Journal of Artificial Intelligence Research*, 23:441–531, 2005.
- [DGLP04] H. E. Dixon, M. L. Ginsberg, E. M. Luks, and A. J. Parkes. Generalizing Boolean satisfiability II: Theory. *Journal of Artificial Intelligence Research*, 22:481–534, 2004.
- [DGP04] H. E. Dixon, M. L. Ginsberg, and A. J. Parkes. Generalizing Boolean satisfiability I: Background and survey of existing work. *Journal of Artificial Intelligence Research*, 21:193–243, 2004.
- [DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.
- [FK03] E. Freudenthal and V. Karamcheti. QTM: Trust management with quantified stochastic attributes. Technical Report TR2003-848, Courant Institute, NYU, 2003.
- [Fre95] J. W. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1995.
- [Fri97] E. Friedgut. Necessary and sufficient conditions for sharp thresholds of graph properties, and the k -SAT problem. Preprint, 1997.
- [GGN⁺03] I. P. Gent, E. Giunchiglia, M. Narizzano, A. G. D. Rowley, and A. Tacchella. Watched data structures for QBF solvers. In *Selected Papers from the Proceedings of the The Sixth International Conference on Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.
- [GHR03] I. P. Gent, H. H. Hoos, A. G. D. Rowley, and K. Smyth. Using stochastic local search to solve quantified Boolean formulae. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (LNCS 2833)*, pages 348–362. Springer-Verlag, 2003.
- [GKM04] M. Ghasemzadeh, V. Klotz, and C. Meinel. Embedding memoization to the semantic tree search for deciding QBFs. In *AI 2004: Advances in Artificial Intelligence*, volume 3339 of *Lecture Notes in Computer Science*, pages 681–693. Springer-Verlag, 2004.
- [GNT02] E. Giunchiglia, M. Narizzano, and A. Tacchella. Learning for quantified Boolean logic satisfiability. In *Proceedings of the Eighteenth*

- National Conference on Artificial Intelligence*, pages 649–654. The AAAI Press/The MIT Press, 2002.
- [GNT03] E. Giunchiglia, M. Narizzano, and A. Tacchella. Backjumping for quantified Boolean logic satisfiability. *Artificial Intelligence Journal*, 145(1-2):99–120, 2003.
- [GNT06] E. Giunchiglia, M. Narizzano, and A. Tacchella. Clause/term resolution and learning in the evaluation of quantified boolean formulas. *Journal of Artificial Intelligence Research*, 26:371–416, 2006.
- [GW98] I. P. Gent and T. Walsh. Beyond NP: the QSAT phase transition. Technical Report APES-05-1998, APES (Algorithms, Problems, and empirical Studies), 1998.
- [GW99] I. P. Gent and T. Walsh. Beyond NP: The QSAT phase transition. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 648–653. The AAAI Press/The MIT Press, 1999.
- [Hoo00] J. N. Hooker. *Logic-Based Methods for Optimization: Combining Optimization and Constraint Satisfaction*. John Wiley & Sons, Inc., New York, NY, 2000.
- [HOTK00] J. N. Hooker, G. Ottosson, E. S. Thorsteinsson, and H.-J. Kim. A scheme for unifying optimization and constraint satisfaction methods. *Knowledge Engineering Review, Special Issue on AI/OR*, 15(1):11–30, 2000.
- [HR05] N. Hristov and A. Remshagen. Local search for quantified Boolean formulas. In *Proceedings of the Forty-Third ACM Southeast Regional Conference*, pages 116–120, 2005.
- [HS05] H. H. Hoos and T. Stützle. *Stochastic Local Search: Foundations and Applications*. Elsevier/Morgan Kaufmann Publishers, 2005.
- [HSAHB99] J. Hoey, R. St-Aubin, A. J. Hu, and C. Boutilier. SPUDD: Stochastic planning using decision diagrams. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, pages 279–288, 1999.
- [HV95] J. N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15(3):359–383, 1995.
- [Jor99] M. I. Jordan. *Learning in Graphical Models*. MIT Press, Cambridge, MA, 1999.
- [JW90] R. G. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.
- [KBKF95] H. Kleine Büning, M. Karpinski, and A. Flögel. Resolution for quantified Boolean formulas. *Information and Computation*, 117(1):12–18, 1995.
- [KKKS98] L. M. Kirousis, E. Kranakis, D. Krizanc, and Y. C. Stamatiou. Approximating the unsatisfiability threshold of random formulas. *Random Structures and Algorithms*, 12(3):253–269, 1998.
- [KMN02] M. J. Kearns, Y. Mansour, and A. Y. Ng. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. *Machine Learning*, 49(2-3):193–208, 2002.
- [KMPS95] A. Kamath, R. Motwani, K. Palem, and P. Spirakis. Tail bounds

- for occupancy and the satisfiability threshold conjecture. *Random Structures and Algorithms*, 7(1):59–80, 1995.
- [KMS96] H. A. Kautz, D. A. McAllester, and B. Selman. Encoding plans in propositional logic. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR-96)*, pages 374–384, 1996.
- [KW99] H. A. Kautz and J. P. Walser. State-space planning by integer optimization. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 526–533. The AAAI Press/The MIT Press, 1999.
- [LGM98] M. L. Littman, J. Goldsmith, and M. Mundhenk. The computational complexity of probabilistic planning. *Journal of Artificial Intelligence Research*, 9:1–36, 1998.
- [Lit97] M. L. Littman. Probabilistic propositional planning: Representations and complexity. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 748–754. AAAI Press / MIT Press, 1997.
- [LMP01] M. L. Littman, S. M. Majercik, and T. Pitassi. Stochastic Boolean satisfiability. *Journal of Automated Reasoning*, 27(3):251–296, 2001.
- [Maj00] S. M. Majercik. *Planning Under Uncertainty via Stochastic Satisfiability*. PhD thesis, Department of Computer Science, Duke University, September 2000.
- [Maj04] S. M. Majercik. Nonchronological backtracking in stochastic Boolean satisfiability. In *In Proceedings of the Sixteenth International Conference on Tools With Artificial Intelligence*, pages 498–507. IEEE Press, 2004.
- [Maj07] S. M. Majercik. APPSSAT: Approximate probabilistic planning using stochastic satisfiability. *International Journal of Approximate Reasoning*, 45(2):402–419, 2007.
- [MB05] S. M. Majercik and B. Boots. DC-SSAT: A divide-and-conquer approach to solving stochastic satisfiability problems efficiently. In *In Proceedings of the Twentieth National Conference on Artificial Intelligence*, pages 416–422. AAAI Press, 2005.
- [MGLA97] M. Mundhenk, J. Goldsmith, C. Lusena, and E. Allender. Encyclopaedia of complexity results for finite-horizon Markov decision process problems. Technical Report UK CS Dept TR 273-97, University of Kentucky, 1997.
- [ML98a] S. M. Majercik and M. L. Littman. MAXPLAN: A new approach to probabilistic planning. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, pages 86–93. AAAI Press, 1998.
- [ML98b] S. M. Majercik and M. L. Littman. Using caching to solve larger probabilistic planning problems. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 954–959. The AAAI Press/The MIT Press, 1998.
- [ML03] S. M. Majercik and M. L. Littman. Contingent planning under uncertainty via stochastic satisfiability. *Artificial Intelligence Journal*,

- 147(1-2):119–162, 2003.
- [MMZ⁺01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Thirty-Ninth Design Automation Conference*, 2001.
- [MR02] S. M. Majercik and A. P. Rusczek. Faster probabilistic planning through more efficient stochastic satisfiability problem encodings. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling*, pages 163–172. AAAI Press, 2002.
- [MSK97] D. A. McAllester, B. Selman, and H. A. Kautz. Evidence for invariants in local search. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 321–326. AAAI Press/The MIT Press, 1997.
- [New03] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003.
- [Nil80] N. J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Company, Palo Alto, CA, 1980.
- [Pap85] C. H. Papadimitriou. Games against nature. *Journal of Computer Systems Science*, 31:288–301, 1985.
- [Pea88] J. Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, San Mateo, CA, second edition, 1988.
- [PT87] C. H. Papadimitriou and J. N. Tsitsiklis. The complexity of Markov decision processes. *Mathematics of Operations Research*, 12(3):441–450, August 1987.
- [PV96] T. J. Park and A. Van Gelder. Partitioning methods for satisfiability testing on large formulas. In *CADE*, pages 748–762, 1996.
- [Rot96] D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence Journal*, 82(1-2):273–302, 1996.
- [SB06] H. Samulowitz and F. Bacchus. Binary clause reasoning in QBF. In A. Biere and C. P. Gomes, editors, *Proceedings of the Ninth International Conference on Theory and Applications of Satisfiability Testing*, volume 4121 of *Lecture Notes in Computer Science*, pages 353–367. Springer, 2006.
- [SBB⁺04] T. Sang, F. Bacchus, P. W. Beame, H. A. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing*, 2004.
- [Sha86] R. D. Shachter. Evaluating influence diagrams. *Operations Research*, 34(6):871–882, 1986.
- [SKC96] B. Selman, H. A. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In D. S. Johnson and M. A. Trick, editors, *Cliques, Coloring, and Satisfiability*, pages 521–531. American Mathematical Society, 1996. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, v. 26.
- [SLM92] B. Selman, H. J. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the National Conference on Artificial Intelligence*, pages 440–446, 1992.

- [SM07] H. Samulowitz and R. Memisevic. Learning to solve QBF. In *Proceedings of the Twenty-Second National Conference on Artificial Intelligence*, pages 255–260. The AAAI Press, 2007.
- [SML96] B. Selman, D. Mitchell, and H. J. Levesque. Generating hard satisfiability problems. *Artificial Intelligence Journal*, 81:17–29, 1996.
- [TMW06] S. A. Tarim, S. Manandhar, and T. Walsh. Stochastic constraint programming: A scenario-based approach. *Constraints*, 11(1):53–80, 2006.
- [Uma99] C. Umans, 1999. Personal communication with M. Littman.
- [VMR99] P. Van Hentenryck, L. Michel, L. Perron, and J.-C. Régin. Constraint programming in opl. In G. Nadathur, editor, *PPDP '99: Proceedings of the International Conference PPDP'99 on Principles and Practice of Declarative Programming*, pages 98–116. Springer-Verlag, 1999.
- [Wal97] J. P. Walser. Solving linear pseudo-Boolean constraint problems with local search. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 269–274. AAAI Press/The MIT Press, 1997.
- [Wal00] T. Walsh. SAT v CSP. In R. Dechter, editor, *Principles and Practice of Constraint Programming—CP 2000*, volume 1894 of *Lecture Notes in Computer Science*, pages 441–456. Springer-Verlag, 2000.
- [Wal01] T. Walsh. Search on high degree graphs. In *IJCAI-01*, pages 266–274, 2001.
- [Wal02] T. Walsh. Stochastic constraint programming. In *Proceedings of the Fifteenth European Conference on Artificial Intelligence*, pages 111–115. IOS Press, 2002.
- [WW00] S. A. Wolfman and D. S. Weld. Combining linear programming and satisfiability solving for resource planning. *Knowledge Engineering Review*, 15(1), 2000.
- [ZKC01] Z. Zeng, P. Kalla, and M. Ciesielski. LPSAT: A unified approach to RTL satisfiability. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 398–402. IEEE Press, 2001.
- [ZM05] Y. Zhuang and S. M. Majercik. WALKSSAT: An approach to solving large stochastic satisfiability problems with limited time. Unpublished: contact smajerci@bowdoin.edu, 2005.
- [ZS00] H. Zhang and M. E. Stickel. Implementing the Davis-Putnam method. *Journal of Automated Reasoning*, 24(1–3):277–296, 2000.

This page intentionally left blank

Subject Index

- NP* parameterized, 683
- SNP* parameterized, 684
- SE*, 683
- g*-means, 492
- k*-nearest neighbour, 492
- #P, 994
- #SAT, 400, 419
- QUANTOR, 399, 402
- Walksat, *see* local search, Walksat
- %RES, 127
- 0-1 integer linear program, 282
- 1UIP
 - literal, 244
 - scheme, 124
- 2-CNF formula, 1139
- 2CNF, 705
- 2QBF solvers, 1018
- 3-CNF, 78
 - conversion to, 236, 261, 279
- 3S, 492–494

- a-numbers, 561
- ablation analysis, 499
- absorb, 252
- absorption, 252
- abstract counterexample, 809
- abstraction, 804
 - Boolean, 1232, 1314
 - conservative, 804
 - existential, 804
 - function, 804
 - lazy, 808, 811
 - predicate, 792, 805
 - refinement, 811
- ACCE, 410, 413
- accessibility relation, 1228

- Ackermann's expansion, 1278, 1310
- acquisition function, 486
- action, *see also* group, action, 767, 1346
 - executable, 767
 - group, 523, 524
 - table, 524
- activation variable, 153
- active
 - effects, 767
 - literal, 1332
 - variables, 411
- acyclic
 - formula, 710
 - graph, 713
- adaptec graph, 530
- adaptive
 - capping, 484–486
 - heuristics, 378
 - ranking, 201
 - restarts, 247, 269
- addition, *see also* cutting planes, 284
 - blocked clause, 397
- additive factor, 1352
- adorning trees, 363
- advanced probing techniques, 393, 407, 408
- affects relation, 775
- affine formula, 672
- AI problems, 937
- AIG, 416, 417
- AIG rewriting, 754
- Alan Robinson, 20
- Alan Turing, 15
- Alasdair Urquhart, 22

- ALE, 407
- Alfred Tarski, 16
- algebraic
 - approaches, 1359
 - methods of reasoning, 270, 310
 - proof system, 270, 311
 - reasoning, 416, 419
 - SAT solving, 281
 - structure, 4
- algorithm
 - approximation, 917, 1350
 - backtracking, 455
 - basic, 50
 - branch and bound, 906
 - branching, 353, 679, 681
 - clause-shortening, 680
 - configuration, 481, 482, 494–496, 498–500
 - problem, 482
 - core-guided, 948
 - cube-covering-based, 677–679
 - decimation based, 882
 - decision
 - MAX-2-SAT, 44
 - MAX-SAT, 44
 - deterministic, 679
 - DP, 104
 - DPLL, 18, 23, 112, 140, 437, 674, 681, 887, 1052
 - classic, 1239
 - fixed-parameter, 694, 695
 - genetic, 485
 - HeerHugo, 111
 - interior point, 45
 - local search, 878
 - message passing, 889, 890
 - model-improving, 946
 - modern MaxSAT, 939
 - myopic, 37
 - non-backtracking, 452
 - portfolio, 969
 - ISAC, 969
 - MaxSAT, 969
 - PPSZ, 675
 - probabilistic, 26
 - probe order backtracking, 36
 - random walk, 451, 675, 676, 878
 - randomized, 679
 - robust, 701
 - SAT, 32, 127
 - Schöning's, 676, 879
 - derandomization, 677
 - schedule, 481, 492, 497
 - selection, 481, 489, 494, 496, 498, 500
 - Structured Procrastination (SP), 488
 - Stålmarck, 109, 1048
 - symbolic SAT, 107
 - tree decomposition, 719
- algorithmic
 - approaches, 221
 - barrier, 445, 454
- alias analysis, 795
- alien subterm, 1304
- all-interval series, 85
- all-SAT, 808
- almost uniform
 - generation, 1019
 - generator, 1019
 - sampling, 1016
- almost unique formula, 578
- Alonzo Church, 15
- analysis
 - ablation, 499
 - alias, 795
 - conflict, 25, 119, 140, 244, 249, 288, 289, 295, 298, 308, 408, 1055, 1106, 1110, 1243
 - parameter importance, 498
- ancients, 6
- and-inverter graph, 416
- annotated clauses, 1192
- answer set programming, 481, 494, 497, 746
- antecedent, 137, 399, 638, 646, 1165
 - clauses, 1187
- anti-Horn, 1138
- application, 933, 936
 - constraint, 672
 - domain, 184
 - proof, 660
- approach
 - algorithmic, 221

- assumption-based, 1314
- automata-theoretic, 1224
- clause-shortening, 680
- CSP-based, 1224
- cube-and-conquer, 645
- DPLL-based, 1224
- eager, 1224, 1256, 1275
- implicit hitting set, 962
- inverse-method, 1224
- lazy, 1285, 1310
- lemma-lifting, 1314
- OBDD-based, 1224
- on-line lazy, 1224
- portfolio, 970
- proof-based, 1314
- tableau-based, 1224
- translational, 1224
- approximation, 1350, 1359
 - algorithm, 917, 1350
 - MaxSAT, 917
 - amplification, 1024
 - multiplicative, 1024
 - clause reduction, 199, 200
 - constant-factor, 1024
 - fixed-parameter, 701
 - model-improving, 968
 - of reachability, 769
 - one-sided, 1019
 - two-sided, 1019
- arc consistency, 945
- Archie Blake, 19
- architecture
 - look-ahead, 185
- Aristotle, 6
- arithmetic
 - bounded, 320, 322
 - mean, 360
 - progression, 833
- Armin Haken, 22
- arrangement, 1305
- array
 - binary implication, 208
 - covering, 835
 - number, 836
 - orthogonal, 836, 837
 - incomplete, 837
- asserting, 245, 289
 - clause, 121, 124, 1186
 - learning scheme, 245
 - literal, 244
- assertion
 - level, 121, 246
 - unwinding, 802
- assertive property, 244
- assignment, 637, 670, 699
 - autarky, 204, 602
 - complete, 135, 550, 1207
 - decision, 137
 - incomplete, 1244
 - necessary, 203
 - partial, 135, 236, 586, 591, 625, 637, 1207, 1292
 - safe, 624
 - satisfiability-preserving partial, 624
 - satisfying, 411, 798
 - total, 236, 637, 699, 1292, 1295
 - satisfying, 586
 - tree, 1332
 - truth, 513, 698, 699, 931, 1049, 1231
 - partial, 1231
 - restricted, 1232
 - restriction, 931
 - total, 699, 1231
- associative operation, 516
- assumption-based SAT solving, 942
- asymmetric
 - covered clause, 410
 - literal, 407–409
 - tableau rule, 407
 - tautology, 244, 407, 408, 410, 1201
 - quantified, 1201
 - resolution, 410
- asymmetric covered clause, 413
- asymmetric tautology, 407
- asymptotic notation, 238
- asymptotically almost surely, 257
- AT, 407
- atom
 - modal, 1231
 - propositional, 1231
 - top-level, 1231
- atomic formula, 1269
 - generalized, 1312

- ATPG, 1064
 - combinational, 1066
 - encoding, 1072
 - experiments, 1077
 - SAT-based, 1070, 1071
 - seealsoautomatic test pattern generation, 1064
- autark
 - assignment, 26
 - sub-clause-set, 589
- autarky, 204, 401, 585, 587, 588, 627, 640
 - assignment, 204, 602
 - autarky system, 605
 - autarky-monoid, 591
 - balanced, 612
 - linear, 613
 - clause, 601
 - closure, 598
 - conditional, 640
 - decomposition, 614
 - detection, 202, 204
 - directly indecomposable, 595
 - duality to resolution, 589
 - finding autarkies, 598
 - lean clause-sets, 589
 - lean kernel, 589
 - linear, 24, 29, 604, 609, 610
 - look-ahead resolvents, 205
 - matching, 608
 - maximal, 593
 - minimal, 594
 - monoid, 591, 593, 596
 - polynomial time, 604
 - principle, 603
 - problem
 - existence, 599, 615, 622
 - non-trivial, 616
 - quasi-maximal, 616
 - pure, 607, 608
 - reasoning, 190, 204
 - reduction, 588, 589, 623
 - semi-maximal, 594
 - semigroup, 593
 - strongly-balanced, 626
 - subsets, 594
 - system, 605, 606
 - trivial, 588
 - weak, 376, 587, 588, 590, 602, 624, 627
 - weakly-balanced, 626
- autarky-resolution duality
 - applications, 600
- AutoFolio, 494, 495
- automata-theoretic approach, 1224
- automatability, 234, 251, 305
- automatable, 251, 282, 305
- automatic test pattern generation, 1047, 1048, 1064, 1066, 1080
- automorphism, *see* group, automorphism
 - graph, 565
 - group, 526
- average-case lower bound, 258, 259, 278
- axiom
 - Boolean, 273
 - clause, 80, 239, 1207
 - frame, 91, 780, 1347
 - generalized, 1190
 - negation, 274
 - nonlogical, 318
 - of choice, 17
 - rule, 1191
 - system, 3
- Böhm solver, 189
- Büchi automaton, 745
- backbone, 40, 151, 193, 417, 472, 1357
 - variables, 1002
- backbone search heuristic (BSH), 191, 193
- backbone search renormalized heuristic (BSRH), 194
- backdoor, 151, 470
 - set, 704
 - deletion, 705
 - strong, 704
 - weak, 705
 - strong, 471
 - sub-solver, 470
 - trees, 705
 - weak, 471
- background, 939

- backjumping, 25, 191, *see also* backtracking, 246, 247, 401, 1055, 1243
 - modal, 1247
 - logics, 1247
 - SMT, 1288, 1291, 1294, 1296
- backtrack search, 465, 475
- backtrack-free search space, 1005
- backtrackability, 1286
- backtracking, 116, 121, 241, 245–247, *see also* branching heuristics, 353, 887, 1164
 - algorithm, 455
 - chronological, 116, 144, 246
 - far, 121
 - non-chronological, 118, 119, 144, 246, 1055, 1343, 1357
 - random problems, 443, 455
 - solver, 352
- backup solver, 490, 491
- backward subsumption, 398–400, 402
- balanced autarky, 612
 - linear, 613
- balanced incomplete block design, 840
- balanced linear satisfiability, 614
- balancedness, 198
- bandwidth auction, 486
- base class, 704
- base translation, 773
- basic
 - algorithm, 50
 - local search method, 218
 - matrix, 574
- BasicILS, 484
- BCE, 409, 410
- BCP, 136, 395, 532
 - implicit, 1000
- BDD, 31, 107, 403, 740, 808, 809, 1048, 1142, 1171
- Begriffsschrift, 12
- behavior
 - fat tailed, 463, 465, 466
 - heavy tailed, 463, 465, 466
- belief propagation, 225, 890, 1008
 - random problems, 454
- Bennett, 821
- Berkmin, 635
- Bertrand Russell, 13
- beyond NP, 159
- biased coin, 1007
- BIG, 403, 404
- big-O notation, 238
- big-O-tilde notation, 307
- bigblue graph, 530
- bijunctive, 1139
- bimander encoding, 81
- binary
 - clause, 207, 373, 392, 396, 403–408
 - preferred, 208
 - decision diagram (BDD), 31, 107, 403, 416, 417, 419, 644, 740, 743, 745, 809, 1048, 1171
 - existential quantification, 31
 - generalized cofactor, 32
 - ordered (OBDD), 108, 383
 - restrict, 32
 - decision tree, 1337
 - encoding, 81
 - format, 651
 - heap, 402
 - implication
 - array, 208
 - graph, 403–408
 - resolution, 126, 127, 405
 - search tree, 696, 703
 - variables, 512, 864
- bioinformatics, 938
- bipartite
 - literal-clause graph, 617
 - variable-clause graph, 617
- BIRD, 1031
- bit predicate, 540
- bit-vector semantics, 791
- bitwise encoding, 81
- bivalence, 1240
- black-and-white principle, 603
- block, 640, 833, 840, 842, 1331
 - existential, 1331
 - partition, 515
 - quantifier, 1178
 - innermost, 1178
 - outermost, 1178
 - randomized, 1331

- universal, 1333
- blocked
 - clause, 92, 93, 393, 394, 409, 410, 413, 417, 418, 625, 640, 682, 1199
 - addition, 397
 - decomposition, 418
 - elimination, 393, 394, 409, 410, 417, 682, 1199
 - globally, 411, 413
 - literal, 1200
- blocking
 - clause, 1023
 - variables, 946
- Bloom filter, 398
- BMC, 739, 792, *see also* bounded model checking, 798
 - encodings, 798
- BMO
 - MaxSAT, 933
- Boole/Shannon expansion, 561
- Boolean
 - abstraction, 1232, 1314
 - algebra, 10, 76
 - axiom, 273
 - circuit, 414, 416, 1048, 1050
 - constrained, 1050
 - satisfiability, 1051
 - conflict, 1288
 - consistency, 522, 532
 - constraint, 672
 - deduction, 1288
 - encoding, 1073
 - gates, 1074, 1075
 - hierarchy, 725
 - hypercube, 304
 - literals, 512
 - multilevel optimization, 933
 - operators, 10
 - optimization problems, 933
 - predicate, 537
 - program, 805, 806
 - refinement, 1232
 - satisfiability, 1064, 1069
 - problem, 562
 - variable, 49, 236, 384, 512, 669, 1088
- Boolean constraint propagation, *see also* BCP, 395, 1056, 1103
 - counter based, 1058
 - implementation techniques, 1057
 - lookup table based, 1058
 - watched literal based, 1059
- Boolean decision diagram, *see also* BDD, 808
- Boolean expression diagram (BED), 1051
- Boolean multilevel optimization (BMO), 933
- bound
 - conflict, 1113
 - lower, 998, 1006, 1113, 1115
 - sphere covering, 677
 - upper, 25, 998, 1009, 1113
 - variable, 1132
- bounded
 - arithmetic, 320, 322
 - heavy-tailed distributions, 469
 - resolution, 671
 - semantics, 742
- bounded model checking, 411, *see also* BMC, 739, 765, 792, 798
 - completeness, 802
- bounded variable addition, *see also* BVA, 403, 650, 651
- bounded variable elimination, *see also* BVE, 392–394, 401–403, 408, 409, 412, 418
- bounded-CNF-SAT, 704
- bounded-depth
 - Frege, *see also* bounded-depth LK, 318
 - bounded arithmetic, 322
 - quasipolynomial size, 322
 - symmetry, 322
 - with parity, 321
 - LK, 319
 - comparison to cutting planes, 321
 - comparison to polynomial calculus, 321
 - comparison to resolution, 321
 - LKoplus, 321
- Box Normal Form (BNF), 1225

- BP, *see* belief propagation
- branch
 - closed, 1236
 - order, 380
- branch and bound, 44, 1338
- branch/merge rule, 111
- branching, 351, 679
 - algorithms, 353, 679, 681
 - conflict-driven, 162
 - evaluating, 354
 - ordering heuristics, 372, 380
 - scheme, 372
 - semantic, 1240
 - syntactic, 1240
 - variable, 372, 380
- branching heuristics, *see also* branching tuples, *see also* distance functions, *see also* measures, 1342, 1356
 - constraint satisfaction problems, 384
 - convexity, 359
 - dsj, 377
 - Franco heuristics, 380
 - general framework, 353
 - Jeroslow-Wang, 1342
 - rule, 377
 - Johnson heuristics, 381
 - justification frontier based, 1063
 - MNC, 375
 - MOM, 377
 - MOMS, 1342
 - non-clausal, 383
 - OKsolver, 375
 - order of branches, 380
 - practical algorithms, 374
 - primary input gate, 1062
 - satisfiability estimator, 373
 - satisfies most clauses, 1342
 - signal correlation based, 1063
 - structure-based, 1062
 - theoretical algorithms, 373
 - top-down, 1063
 - tree size, 362
 - two-sided Jeroslow-Wang rule, 377
- branching randomization, 146
- branching strategy, 384
- branching tuples, 357, 362, 368, *see also* projection functions
 - canonical order, 368
 - composition, 358
 - concatenation, 358
 - means, 360
 - probability distribution, 362
 - product rule, 370
 - tau-function, 359
- branchwidth, 718
- breadth first search (BFS), 753
- break count, 216
- broken mosquito screen formula, 305
- bucket, 105
 - elimination, 105, 1349
- bug finding, 482, 499
- BVA, 403
- BVE, 392
- byte encoding, 651
- caching, 999, 1344
 - component, 999, 1344, 1349, 1358
 - formula, 999, 1344
 - modal logics, 1245
- CaDiCaL, 394, 400, 409, 413
- calculus
 - polynomial, 234, 272, 273, 275, 279, 322, 1214
 - space, 279
 - Q-resolution, 1179
 - ratiocinator, 10
- call stack, 792
- cancellative groupoid, 821
- cancelling
 - addition, 285
 - linear combination, 285
- candidate, 1194, 1196
- canonical
 - decision tree, 1337
 - labeling, 565
 - projection, 357
- capping
 - adaptive, 484–486
 - dynamic, 488
- CapsAndRuns, 489
- cardinality constraint, 81, 92, 156, 284, 743, 944, 1089

- arc consistency, 945
- atleast, 1089
- atmost, 1089
- conditional, 157
- detection, 310
- encoding, 156
- exactly, 1089
- MaxSAT, 944
- negative, 259
- positive, 259
- reduction rule, 296
- soft, 157
- cardinality minimal subsets, 158
- cardinality reasoning, 251
- cardinality-minimal subsets, 158
- categorical
 - logic, 6
 - propositions, 6
- cavity method, 224, 873
- Cayley's theorem, 521
- CBMC, 802
- CC-balanced expressions, 28
- CCE, 410, 413
- CDCL, 140, 161, 163, 234, *see also*
 - conflict-driven clause learning, 323, 391–393, 395, 397–401, 408, 415, 418, 500, 644, 1177, 1204
 - clause deletion, 148
 - conflict analysis, 244, 249, 289, 298
 - conflict-driven branching, 149
 - heuristics, 247, 254, 268
 - lazy data structures, 149
 - phase saving, 148
 - proof logging, 249
 - SAT, 133, 152
 - solvers, 133, 140, 151, 152, 155, 160, 161, 163, 418
 - search restarts, 146
 - solvers, 392, 393, 397, 415, 500, 644
 - using extended resolution, 313
- CDCL(XOR), 1031
- cell
 - mapped, 551
 - partition, 515
 - target, 528, 551
- certifying function, 708
- CFG, *see* control flow graph
- chain
 - formula, 1038
 - resolution, 638, 646
- chaining variable, 543
- chance constraints, 1354
- channelling clauses, 81
- Charles Sanders Peirce, 16
- Chebyshev inequality, 1023
- chronological backtracking, 116, 144, 246
- Chvátal-Gomory cut, *see also* division, 285
- circuit, 416, 417, 1048
 - Boolean, 416, 1048, 1050
 - constrained, 1050
 - complexity, 320, 1214
 - monotone, 305
 - encoding, 744
 - industrial, 1079
 - monotone, 305
 - representation, 1048
 - satisfiability
 - checking, 1052
 - problem, 1051, 1052
 - synthesis, 417
- circuit-level
 - reasoning, 416
 - rewriting, 417
- clasp, 487
- claspfolio, 494
- class
 - base, 704
 - complexity
 - uniform, 687
 - easy, 671
 - explanations, 11
 - heterogeneous base, 709
 - intractable, 671
 - non-uniform, 687
 - of easy expressions, 27
 - of matched expressions, 30
 - parallel, 840
 - partition, 515
 - q-Horn, 29

- qualitative, 618
 - tractable, 671
- classes of satisfiability
 - 2-SAT, 27
 - 3-SAT, 1345
 - CC-balanced, 28
 - Horn, 28
 - k-SAT, 1336
 - matched, 30
 - minimally unsatisfiable, 30
 - nested, 30
 - renameable Horn, 28
 - single lookahead unit resolution, 29, 395
- classic
 - DPLL algorithm, 1239
 - DPLL procedure, 1238, 1240
- classical
 - planning, 766
 - symmetry, 562
- classification, 490
 - pairwise, 491
- Claude Shannon, 18
- clausal
 - core, 411
 - proof, 411, 635
- clause, 75, 101, 236, *see also* implicate, 514, 637, 669, 698, 766, 1015, 1090, 1146, 1178
 - k-clause, 236
 - addition
 - rules, 1199
 - annotated, 1192
 - antecedent, 1187
 - asserting, 121, 124, 1186
 - asymmetric covered, 410
 - at-least-one, 78
 - at-most-one, 78
 - autarky, 601
 - axiom, 80, 239, 1207
 - binary, 207, 373, 392, 396, 403–408
 - efficient storage, 207
 - blocked, 92, 93, 393, 394, 409, 410, 413, 417, 418, 625, 640, 682, 1199
 - addition, 397
 - decomposition, 418
 - elimination, 393, 394, 409, 410, 417, 682, 1199
 - globally, 411, 413
 - blocking, 1023
 - channelling, 81
 - clash, 698
 - cloning, 951, 957
 - conditional autarky, 640
 - conflict, 78, 119, 1055
 - conflict-driven, 119–121
 - deletion, 125
 - conflict-induced, 1107
 - conjunction, 75, 571, 582, 931
 - contradictory, 1161
 - covered, 410, 411
 - critical, 674
 - definition, 393, 639
 - deletion, 249, 394, 643
 - distribution, 401
 - empty, 102, 185, 395, 397, 411, 586
 - detection, 712
 - equivalence, 414
 - forgetting, 394
 - globally-blocked, 641
 - hard, 905
 - hard clauses in MaxSAT, 931
 - implied, 84, 87, 91–93
 - induced, 705
 - ladder validity, 81
 - learned, 1055
 - learning, 23, 25, 121, 140, 269, 392, 394, 398, 399, 472, 477
 - length, 669
 - look-ahead autarky, 625
 - maximality, 80
 - modification rules, 1200
 - non-relaxable, 905
 - number, 37
 - overlap, 698
 - parent, 590
 - positive, 36
 - propagation-redundant, 410
 - re-weighting, 218
 - dynamic, 221

- reduction
 - approximation, 199, 200
 - heuristics, 190, 193
- redundant, 403, 637
- relaxable, 905
- resolution-subsumed, 410
- satisfiability-preserving, 624
- set-blocked, 410, 641
- sets, 586
 - finite, 598
 - independent, 577
- single, 614
- soft, 905
- soft clauses in Maxsat, 931
- straddle, 30
- subsumption, 393
- support, 79
- tautological, 407
- unit, 20, 103, 185, 395, 637, 670, 883, 1339
 - elimination, 670
- unsatisfied, 217
- vertices, 532
- width, 23, 33
- clause deletion, 125
- clause density, 41, 670, 682
 - constant, 682
 - upper bound, 682, 685
- clause distribution, 401, 402
- clause elimination, 1199
 - rules, 1199
- clause erasure, *see* clause deletion
- clause forgetting, 394, 407
- clause induced, 705
- clause learning, 23, 25, 140, 241, 243, 391, 392, 394, 398–400, 408, 409, 472, 477, 1055, 1166
 - antecedent, 137
 - chronological backtracking, 144
 - clause minimization, 144
 - conflict analysis, 140
 - decision level, 137
 - first unique implication point (1UIP), 244
 - implication graph, 138
 - non-chronological backtracking, 143
 - unique implication point (UIP), 244
 - unique implication points, 143
 - clause minimization, 255, 399
 - clause re-weighting, *see* clause weighting
 - clause reduction, 407
 - clause redundancy, 418
 - clause shortening, 679
 - clause space, *see also* space, 262, 274
 - clause variable incidence graph (CVIG), 971
 - clause weighting, 218, 219, 221, 931
 - clause-minimal formula, 580
 - clause-preserving homomorphism, 577
 - clause-set, 586
 - barely lean, 614
 - complement-invariant, 579, 612
 - deficiency, 572
 - empty, 586
 - formal, 607
 - hitting clause-sets, 577
 - irredundant, 580
 - labelled, 611
 - lean, 589
 - literal-regular, 617
 - matched, 608
 - matching satisfiable, 608
 - minimally unsatisfiable, 571
 - marginal, 578
 - saturated, 578
 - notations, 371
 - uniform, 617
 - unsatisfiable
 - MaxSAT, 932
 - variable-regular, 617
 - clause-shortening
 - algorithm, 680
 - approach, 680
 - clause-to-variable ratio, 222
 - clause-variable matrix, 611
 - clique formula, 258, 259
 - clique-coclique formula, 304, 321, 322
 - weak, 321
 - clique-colouring formula, *see also* clique-coclique formula, 304, 315
 - clique-width, 720

- closed
 - branch, 1236
 - formula, 1132
- closure, 1252
 - autarky, 598
 - congruence, 1273
- cluster formula, 711
- clustering, 490, 492, 493
 - phenomenon, 869
- CNF, 33, 75, 101, *see also* conjunctive normal form
 - DIMACS format, 82
 - encoding, 86, 91, 946
 - extensional constraints, 78
 - formula, 669, 698, 1150, 1270
 - matched, 724
 - preprocessing, 409
 - symmetry, 525
 - transformation, 75
 - translation to polynomials, 271
 - translation to pseudo-Boolean constraints, 282
 - trivial, 104
- CNF formula, 236, 316, 698
 - isomorphic, 698
 - k -CNF formula, 236
 - length, 698
 - lifted, 301
 - model, 699
 - random k -CNF, 257, 262, 277, 279, 305, 312
 - renaming, 698
 - restriction, 699
 - satisfiable, 699
 - size, 698
 - unsatisfiable, 699
 - minimal, 725
- CNF-model, 584
- CNFuzzDD, 489
- coarse threshold, 40
- codewords, 677
- coefficient of variation, 1024, 1026
- cofactor, 561, 751
- coherence, 1230
- coherent, 1230
- cohesion, 580
- coloring, *see* ordered partition, 527
 - stable, 527
- colourability, 258
- colouring, *see also* clique-colouring, 277, *see also* even colouring, 282
 - graph, 79
- colouring formula, 258, 277, 278
- combination, 493, 1299
 - method, 1306
 - precise, 375
- combinatorial
 - design problems, 821
 - problems, 695, 937
- COMiniSatPS, 402
- commander
 - encoding, 81
 - variables, 81
- common subexpression elimination, 416
- communication complexity, 301, 306
- community
 - graph, 723
 - structure, 723, 1357
- commutatively
 - generating set, 596
 - semi-generating set, 596
- compilation, *see* knowledge compilation
- compiler, 795
- complement, 637, 766
- complement-invariant
 - clause-sets, 579, 612
 - formula, 579
 - formulas, 579
- complete, 1234
 - assignment, 135, 550, 1207
 - set, 1237
- completeness, 237, 240, 576, 747, 753, 814, 994, 1295, 1306
 - of first-order logic, 17
 - refutation, 102, 1052
 - theory, 694
 - threshold, 748, 802
- complexity, 365, 1094, 1334
 - communication, 301, 306
 - computational, 1137, 1158, 1334
 - parameterized, 694

- problems, 354
 - absolute, 371
 - relative, 371
- proof, 121, 234, 237, 442
 - propositional, 726
- resolution, 21, 23, 443
- results, 1131, 1137, 1334
- worst-case, 893
- complexity classes
 - #P, 1017
 - FPRAS, 1017, 1024
 - polynomial hierarchy, 1017
 - probabilistic, 1017
 - uniform, 687
- component, 998, 1031
 - connected, 394, 400, 404, 999
 - disconnected, 400
- component caching, 400, *see also* caching, 999, 1344, 1349, 1358
- computable function, 15
- computation tree logic, 740
- computational complexity, 1137, 1158, 1334
 - NP^{PP} , 1334
 - #P, 1334
 - NP, 1334
 - PP, 1334
 - PSPACE, 1334
- computational complexity theory, 322
- concatenation, 358
- concretization function, 804
- condensation transition, 446
- conditional
 - autarky, 640
 - part, 640
 - symmetry, 558
- conditional cardinality constraints, 157
- conditioning, 103
- cone, 358
- cone of influence, 1051
- cone-of-influence reduction, 393, 409
- confidence, 1018, 1020
 - correctness, 1003
- Configurable SAT Solver Challenges, 487, 495, 500
- configuration, 936
 - algorithm, 481, 482, 494–496, 498–500
 - problem, 482
- conflict, 136, 243, 395–397, 408, 638, 1168, 1288
 - analysis, *see* conflict analysis
 - Boolean, 1288
 - bound, 1113
 - clause, 78, 119
 - graph, 244, 617, 700
 - mixed Boolean+theory, 1288
 - partition, 1109
 - set
 - generation, 1286
 - modal, 1247
 - state, 553
 - theory, 1286, 1288, 1291, 1293, 1296, 1308, 1309
 - treewidth, 718
- conflict analysis, 25, 119, 140, 244, 249, 288, 289, 295, 298, 308, 408, 1055, 1106, 1110, 1243
- CDCL, 244, 249, 289
 - comparison of CDCL and pseudo-Boolean solving, 292
- invariant, 289
- pseudo-Boolean, 288, 295, 308
- reduction algorithm, 291, 294, 295
 - using division, 293
 - using saturation, 290
- conflict clause
 - minimization, 636
- conflict driven learning, 1055
- conflict set, 118, 120
- conflict-driven
 - branching, 162
 - clause, 119–121
 - deletion, 125
 - clause learning, 161, *see also* CDCL, 234, 241, 323
 - pseudo-Boolean solving, *see* pseudo-Boolean, solving
 - strategy, 183
- conflict-induced clause, 1107
- conflict-regularity, 618
- conflicting constraint, 288

- congruence closure, 1273
- congruence constraints, 1290
- conjugate
 - equivalent, 826
 - implicant, 826, 830
 - Latin square, 826
 - orthogonality, 827
- conjunction
 - clauses, 75, 571, 582, 931
 - finite, 637
 - literals, 1304
 - logical, 512
- conjunctive normal form (CNF), 75, 101, 135, 233, 236, 514, 571, 637, 669, 698, 1015, 1047, 1157, 1160, 1178
 - comparison to pseudo-Boolean constraints, 283
 - quantified Boolean, 1135
- connected component, 394, 400, 999
- consensus, 19
 - graph, 700
 - treewidth, 718
- consequence, 1133
- conservative abstraction, 804
- consistency, 3, 4, 92
 - arc, 945
 - Boolean, 522, 532
 - edge, 532
- constant width distribution, 33
- constant-factor approximation, 1024
- constraint, 78, 383, 384, 1138, 1354
 - application, 672
 - at-most-one, 81
 - binary, 78
 - Boolean, 672
 - cardinality, 81, 82, 92, 418, 743, 944, 1089
 - MaxSAT, 944
 - chance, 1354
 - congruence, 1290
 - difference, 1281, 1282
 - equivalence, 1290
 - fairness, 745, 747
 - inequality, 82
 - intensional, 78, 81
 - parity, 82, 90
 - pseudo-Boolean, 157, 944
 - quantified, 1138
 - resolvent, 203
 - simple path, 748
 - sum, 1281
 - symmetry, 564
 - transitivity, 1283, 1284, 1290
 - translation, 1118
 - unit, 1103
 - XOR, 414
- constraint programming, 299, 310, 395
- constraint resolvents, 203, 204
- constraint satisfaction, 481, 819
 - stochastic, 1354
- constraint satisfaction problem (CSP), 78, 118, 384, 559, 862, 938
- contingency variables, 782
- contingent planning, 781, 782
- contradictory clause, 1161
- control flow, 799
- control flow graph, 800
- Cook translation, 322
- core, 756
 - clausal, 411
 - MaxSAT, 932
 - unsatisfiable, 152, 154, 601, 659, 812, 1314
- core-guided algorithms, 948
- correctness confidence, 1003
- coset, 518
 - left, 518
 - pruning, 554
 - right, 518
- cost, 1213, 1214
- cost function, 1092
 - MaxSAT, 929
- cost-sensitive decision forest, 491
- cost-sensitive hierarchical clustering (CSHC), 493
- count
 - expected total model, 1006
 - saturated, 1021
- counterexample, 795, 798
 - abstract, 809
 - spurious, 811
- counterexample guided abstraction refinement (CEGAR), 805, 1194

- countermove, 1194
- counting
 - approximate, 994
 - exact, 994
 - problem, 994
 - reduction, 995
 - weighted, 1035
- counting satisfying assignments
 - belief propagation, 382
 - survey propagation, 382
- counting-mod- p principle, 319, 320
- cover, 225
- coverage, 219, 1034
- covered clause, 410, 411, 413
- covered clause elimination (CCE), 410
- covered literal, 410
- covering, 677
 - array, 835
 - array number, 836
 - code, 677
 - cube, 674, 677
 - products, 717
- CP, *see* cutting planes
- Craig interpolation, *see also* interpolation, 305, 321
- critical clauses, 674
- critical satisfiable formulas, 571
- cross-validation, 494
- CryptoMiniSat, 415, 1016, 1031
- CSHC, 493
- CSP-based approach, 1224
- CTL, 740, 798
- cube, 1015, 1033, 1166
- cube covering, 674, 677
- cube-and-conquer, 645
- cumulative space, 266, 268
- cut, 1109
 - Chvátal-Gomory, 285
 - enumeration, 417
 - fractional, 1117
 - mixed-integer, 1118
 - rule, 317, 914
 - variables, 1031
- cut-set, 1349
- cutting plane proof system, 1096
 - cutting plane, 1110
 - cutting planes, 234, 235, 282, 300, 1116, 1214
 - addition, 284
 - cancelling linear combination, 285
 - coefficient size, 303
 - counting, 302
 - CPstar, 303, 304, 307
 - division, 284, 308
 - general, 286, 304, 307
 - general division, 286
 - generalized resolution rule, 308
 - implicational completeness, 293, 308
 - incomparable to polynomial calculus, 303
 - length, 302
 - line space, *see also* cutting planes, space, *see also* space, 302
 - literal axiom, 284
 - multiplication, 284
 - polynomial-magnitude coefficients, 303, 304, 307
 - reduction to monotone circuits, 305
 - saturation, 286, 308
 - separation
 - from polynomial calculus, 303
 - from resolution, 302
 - simulation of resolution, 302
 - size, 302
 - size-space trade-off, 306, 307
 - space, 304
 - symmetry, 322
 - total space, 304
 - weakening, 286
 - with division, 309
 - with resolution, 287, 307
 - with saturation, 287, 307, 309, 311
 - with saturation and resolution, 287, 307
- cutting plane proof system, *see* cutting planes
- cycle
 - disjoint, 520
 - Eulerian, 278
 - length, 521

- notation, 520
- phase-shift, 539
- structure, 540, 560
- cyclic
 - k -tuple, 842
 - group, 519
 - subgroup, 519
 - zipper method, 834
- d-decomposable negation normal form (DNNF), 1001
- data analysis problems, 937
- data structures, 913, 1356
 - lazy, 149, 1104, 1356
- Davis-Logemann-Loveland procedure, 887
 - modal logics, 1231
 - non-normal logics, 1249
 - SMT, 1286
- Davis-Putnam procedure, 104, 401, 1191
- Davis-Putnam-Logemann-Loveland procedure, 887, 1236, 1338
- DDPP, 831
- deciding satisfiability, 714
- decimation based algorithms, 882
- decision, 119
 - algorithms
 - MAX-2-SAT, 44
 - MAX-SAT, 44
 - assignments, 137
 - diagram, 403, 419, 1163, 1171
 - modal logics, 1252
 - heuristic, 186
 - level, 137, 243, 1167
 - list, 1205
 - literal, 242
 - problem, 615
 - stack, 137
 - state, 553
 - step, 550
 - strategy, 247, 254
 - tree, 493, 1337, 1338
 - binary, 1337
 - variables, 1354
- decoder, 821
- decomposability, 1001
- decomposable negation normal form, 1001
- decomposition
 - blocked clause, 418
 - chart, 561
 - tree, 713, 714
 - algorithms, 719
- deduction
 - Boolean, 1288
 - interface equalities, 1286
 - literals, 1288
 - unassigned, 1286
 - theory, 1288
- deficiency, 572, 580, 613, 618, 725
 - clause-sets, 572
 - cohesion, 580
 - maximal, 572
 - maximum, 725
 - reduced, 620
 - maximal, 623
- definability, 1026
- definition clause, 393, 639
- degree, 272, 274, 283, 723, 1088
 - Nullstellensatz, 272
 - of a literal, 617
 - of a variable, 617
 - of falsity, 283
 - polynomial, 274
 - polynomial calculus, 274
 - v.s. size, 276
 - v.s. space, 280, 281
- delayed cut variable binding, 1349
- delayed theory combination (DTC), 1292, 1308
- deletion, 705
 - backdoor set, 705
 - clause, 125, 249, 394, 643
 - variables, 705
- delta-debugging, 654, 660
- density, 184, 1004
 - clause, 41, 670, 682
 - solution, 91, 93
- density of solutions, 1033
- Dependency Quantified Boolean Formulas (DQBF), 1145
- dependency scheme, 1182, 1213
 - standard, 1182

- trivial, 1182
- depth, 219, 362, 1225
 - formula, 318
 - jump, 198
 - lower bound, 320
- depth- d
 - Frege, 319
 - LK lower bound, 319
 - LK proof, 319
- derivability, 3
- description logics, 1230
- design, 272
 - perfect Mendelsohn, 842
- design choices, 481, 494
- detection
 - autarky, 202, 204
 - empty clause, 712
 - group, 967
 - loop, 814
 - of CNF symmetries, 531
 - pure literal, 1136
- determinism, 1002
- deterministic
 - additive approximate model counter, 1018
 - algorithm, 679
 - heuristics, 475
 - randomization, 476, 477
- deterministic, decomposable negation
 - normal form (d-DNNF), 1001
- diagonally ordered magic square, 844
- diameter, 184, 748
- Dichotomy theorem, 1139
- difference constraints, 1281, 1282
- difference gradient, 220
- difference heuristics, 186, 192
- difference logic, 1273, 1281, 1284, 1285, 1292, 1293, 1307, 1315
- dilemma rule, 110
- DIMACS, 647
 - format, 82
- direct encoding of theory axioms, 1283
- direct implications, 202
- directed acyclic graph (DAG), 239, 263
- direction heuristic, 192, 195
- directional resolution, 105, 1349
- directly indecomposable autarky, 595
- disconnected component, 400
- discrete integration, 1036
- discrete Lagrangian method, 219
- discrete partition, 515
- disjoint cycle, 520
- disjoint DNF, *see* clause-sets, hitting
 - clause-sets
- disjoint quasigroups, 831
- disjunction, 1307
 - logical, 512
- disjunctive
 - formula, 578
 - splitting, 574
- disjunctive normal form, 514, 1015
- disjunctive subformulas, 1240
- dispersion index, 1024, 1025
- distance, 366
 - function, 365, 372, 378
 - optimisation, 378
- Hamming, 677
- heuristic, 186
- penalty, 222
- distillation, 393, 406, 408, 409, 412
- distribution
 - bounded heavy-tailed, 469
 - clause, 401, 402
 - fat-tailed, 466
 - heavy-tailed, 126, 465, 466
 - leptokurtic, 466
 - stable, 466
 - uniform, 1005
- divide and conquer, 1335, 1348
- division, *see also* cutting planes, 284, 287, 308
 - general, 286
- DLL, 395
- DLM, *see* discrete Lagrangian method
- DNF, *see* disjunctive normal form
- DNF width, 1033
- DNF-XOR, 1034
- domain, 1230
 - application, 184
- dominance, 702
- dominate vertex, 1108
- domination, 702
 - equivalent, 702

- incomparable, 702
- strict, 702
- dominator, 143
- don't care
 - observability, 1060
- Donald Loveland, 20
- double look-ahead, 202, 205, 206
 - heuristics, 206
 - resolvents, 207
- download assignment, 1207
- DP, 401
 - algorithm, 104
- DPLL, 112, 140, 161, 241, 323, 395, 401, 707, 718, 725, 754, 887, 997, 1052, 1055, 1338
 - algorithm, 18, 23, 112, 140, 437, 674, 681, 887, 1052
 - classic, 1239
 - framework, 23, 185
 - modal, 1231, 1238, 1239, 1242
 - procedure
 - classic, 1238, 1240
 - with clause learning (DPLL-CL), 245
- DPLL-based approach, 1224
- DPLL-style, 994
- DRAT, 235, 299, 313, 410, 411, 416
 - proof system, 313
 - symmetry reasoning, 323
 - without new variables, 315
- dtree, 1002
- dual
 - graph, 700
 - hypergraph, 700
 - parameter, 704
 - treewidth, 713
- dual-Horn formula, 672
- dual-rail MaxSAT, 323
- duality, 589
- duration, 780
- dynamic capping, 488
- dynamic clause re-weighting, 221
- dynamical transition, 446

- E-MAJSAT, 1334
- eager approach, 1224, 1256, 1275
 - box lifting, 1259
 - modal logics, 1256
 - on-the-fly BCP, 1260
 - on-the-fly pure-literal-reduction, 1261
 - to modal reasoning, 1256
 - to SMT, 1268, 1275, 1310
- eager data-structures, 189, 207
- early pruning, 1290
 - modal logics, 1244
 - SMT, 1288, 1290
 - enhanced, 1291
- easy-hard-easy pattern, 465
- EC formula, *see* even colouring formula
- eccentricity, 748
- edge capacity, 534
- effect precondition, 767
- effective process, 9, 14
- effective weight function, 1036
- effects, 767
 - active, 767
- eigenvalues, 39
- electronic design automation, 739, 756
- elementary magic square, 844
- elementary symmetric function, 561
- elimination
 - asymmetric literal, 407
 - blocked clause, 393, 394, 410, 682
 - bucket, 105, 1349
 - common subexpression, 416
 - Gaussian, 415, 416, 644
 - hidden literal, 393
 - non-shared-input, 409
 - pure literal, 699, 702, 707, 1339
 - pure variable, 1338
 - tautology, 402
 - unit clause, 670, 681
 - variables, 90, 392, 394, 401–403, 408, 409, 412, 418
 - bounded, 393, 401, 402
- embedded Latin square, 823
- Emil Post, 15
- empirical performance model, 485, 491
- empty clause, 102, 185, 395, 397, 411, 586
- empty clause detection, 712
- empty clause-set, 586

- empty literal, 1332
- encoder, 821
- encoding, 75, 391, 392, 403, 414, 416–418, 1072, 1078, 1159, 1347
 - bimander, 81
 - binary, 81
 - binary transform, 80, 93
 - bitwise, 81
 - BMC, 798
 - Boolean, 1073
 - byte, 651
 - cardinality constraints, 156
 - circuit, 417, 744
 - CNF, 86, 91, 946
 - commander, 81
 - direct, 78, 92
 - ladder, 81
 - relaxed, 81
 - linear, 745
 - literals, 653
 - log, 79, 92, 93
 - minimal, 824
 - original, 744
 - pairwise, 81
 - propositional, 743
 - pseudo-boolean constraints, 157
 - size, 91, 944
 - small-domain, 1280
 - sparse, 78
 - support, 79, 92
 - Tseitin, 76, 91, 93, 416, 417, 797
 - variable-byte, 652
 - various approach, 929
- enforcement, 937
- entries, 823
- enumeration
 - cut, 417
 - set, 159
 - trees, 355
- EqSatz, 416
- equality formula, 1213
- equality reduction, 127, 1001
- equality variable, 537
- equisatisfiability, 313, 699
- equisatisfiable, 392, 394, 395, 397, 409, 410, 638, 699, 1158
- equivalence, 1230
 - checking, 739
 - clause, 414
 - constraints, 1290
 - model, 1144
 - reasoning, 190
 - reduction, 393, 404
 - replacement, 1200
- equivalent, 909, 1158, 1230
 - domination, 702
 - literal, 393, 396, 404, 406, 416, 417
 - quasigroups, 849
- equivalent-literal substitution, 393, 404, 412, 413
- ER, *see* extended resolution
- Erdős-Rényi random graph, 258
 - edge density, 258
- error locations, 795
- error state, 795
- essential prime implicants, 19
- estimate
 - quality, 1003
 - without any guarantees, 994
- estimating tree sizes, 362
- estimator
 - 0-1, 1033
 - Franco, 380
 - Johnson, 380, 381
 - satisfiability, 372, 380
 - unbiased, 1020, 1021, 1033
- ETH, 396, 683, 686, *see also* exponential time hypothesis
- Euler conjecture, 824
- Eulerian cycle, 278
- evaluating branching, 354
- evaluation, 1132
- evaluation projection, 357
- Evans, 830
- even colouring formula, 278, 279, 310, 311
 - pseudo-Boolean form, 310
- even number, 1007
- exact counting, 994
- exceptionally hard instances, 465
- exclusive OR, 515
- existential
 - abstraction, 804

- block, 1331
- literal, 1161, 1332
- player, 1178
- quantification, 104, 403, 1331
- variable, 1331
- expander graph, 257, 279, 320
- expansion, 1193
 - Ackermann's, 1278, 1310
 - Boole/Shannon, 561
 - partial, 1193
 - Shannon, 18
 - universal variable, 1361
- expansion-based solving, 1183, 1190–1198
- expected total model count, 1006
- experiments, 268, 269, 310
- exploitation, 486
- exploiting runtime variation, 472
- exploration, 486
- exponential, 437
 - regime, 887, 889
- exponential time hypothesis, 396, 683, 697
- exponentially many regions, 446, 449
- expression
 - CC-balanced, 28
 - minimally unsatisfiable, 30
 - quantified constraint, 1138
 - treewidth, 719
- expression treewidth, 718
- EXPTIME-complete, 268
- extend solution, 413
- extended Frege
 - proof, 317, 318
 - system, 316, 322
 - bounded arithmetic, 322
- extended resolution, 20, 234, 235, 251, 312, 409, 639, 651
 - derivation, 313
- extension
 - axiom, 317
 - function, 1230
 - rule, 312, 315, 639
 - variable, 283, 287, 317
- extensional constraints in CNF, 78
- extensional interpretation, 9
- external variables, 412
- extraction
 - gate, 417
 - strategy, 1211
- extremal properties, 269
- extremal set, 399
- factor
 - additive, 1352
 - graph, 1360
- factoring, 769
- failed literal, 187, 188, 395–397, 405, 406, 408, 411, 1000
- failed-literal probing, 395
- failure, 883
- fairness constraints, 745, 747
- FALCON, 831, 849
- falsified node, 25
- falsify, 237
- fan-in, 1049
- fan-out, 1049
- far-backtracking, 121
- fat tailed behavior, 463, 465, 466
- fat-tailed distribution, 465, 466
- fault, 1065
 - redundant, 1067
 - testable, 1067
- fault model, 1065, 1066
 - stuck-at, 1066
- fault shadow, 1071
- feasible interpolation, 1210
- feature computation, 491, 495, 497
- Fibonacci numbers, 356
- field, 270
 - finite, 415
 - Galois, 415
- Field Programmable Gate Array (FPGA), 1159
- FINDER, 831
- finite conjunction, 637
- finite field, 415
- finite size scaling, 223, 863, 877
- first unique implication point (1UIP), 244, 1180
 - clause, 244
- first-order logic, 399, 419, 1215
 - completeness, 17
- first-order symmetry, 562

- fixed literal, 411
- fixed point, 809
- fixed-parameter
 - algorithm, 694, 695
 - approximation, 701
 - complexity, 393
 - intractability, 697
 - tractability, 694, 695
- fixpoint, 746, 747, 753, 754
- FLoC Olympic Games, 487, 495
- focused random walk, 215
- FocusedILS, 484, 486
- forall reduction, 1136, 1361
- ForallExp-Res, 1177, 1190–1198, 1207, 1212
 - function extraction, 1206
- forced variable, 186
- form
 - prenex, 1159
 - normal, 1134
 - smoothed, 1002
- formal
 - clause-set, 607
 - variables, 611
- format
 - binary, 651
 - proof, 645
 - with hints, 645
 - with witnesses, 650
 - without hints, 649
- formula, 4, 637, 669, 698
 - 2-CNF, 1139
 - acyclic, 710
 - affine, 672
 - almost unique, 578
 - atomic, 1269
 - generalized, 1312
 - broken mosquito screen, 305
 - caching, 1344
 - chain, 1038
 - clause-minimal, 580
 - clique, 258, 259
 - clique-coclique, 304, 321, 322
 - clique-colouring, 304, 315
 - closed, 1132
 - cluster, 711
 - CNF, 236, 316, 669, 698, 1270
 - isomorphic, 698
 - matched, 724
 - renaming, 698
 - satisfiable, 699
 - unsatisfiable, 699
- colouring, 258, 277, 278
- complement-invariant, 579
- counting-mod- p principle, 319, 320
- critical satisfiable, 571
- depth, 318
- disjunctive, 578
- dual-Horn, 672
- equality, 1213
- even colouring, 278, 279, 310, 311
- functional pigeonhole principle, 277
 - ground, 1269
- hitting, 577, 711
- Horn, 264, 672
 - quantified extended, 1148
- house-sitting principle, 272
- inconsistent, 1133
- independent set, 258
- induction principle, 272
- labeled, 1228, 1257
- least number principle, 261
- lifted, 301
- logically equivalent, 638, 1133
- marginal, 578
- matching principle, 272
- maximal, 578
- MaxSAT, 931
 - partial, 932
 - unweighted, 932
- minimal false, 583
- minimal unsatisfiable, 725
- nested, 30
- non-clausal, 580
- ordering principle, 261, 276
- parity, 416, 1210
- parity principle, 319
- pebbling contradiction, 263, 264, 272, 276, 280, 281
- pigeonhole principle, 256, 277, 279, 302, 304, 310, 313, 315, 318–322

- prime implicate, 1139
- propositional, 316
- random k -CNF, 257, 277, 279, 305, 312
- residual, 442
- satisfiability equivalent, 1133
- satisfiable, 159, 637, 775, 889, 1133
- s-gen formula, 259
- stable, 583
- subset cardinality, 259, 277, 309, 310
- tiling problem, 258
- treeSAT, 1353
- trivially satisfiable, 671
- Tseitin, 256, 262, 275, 276, 279, 281, 305, 315, 320–322
- unique minimal unsatisfiable, 578
- unsatisfiable, 637, 887, 1133
- vertex cover, 258
- XOR-CNF, 1139
- XOR-ification, 264
- XOR-substitution, 264
- zero-one design, 259
- formula caching, *see* caching
- forward subsumption, 397, 399, 400
- FPRAS, 996, 1032
- FPT, 695
- fpt-reduction, 697
- fractional cut, 1117
- frame axiom, 91, 780, 1347
- frame Latin square, 828
- framework
 - DPLL, 23, 185
 - labeled CNF, 967
 - statistical, 1009
- Franco estimator, 380
- Franco heuristics, 380
- free binary decision diagrams (FBDD), 54, 1140
- free variable, 1132
- freebie rule, 217
- Frege
 - extended, *see* extended Frege proof, 316, 317
 - system, 235, 316, 322, 1214
 - bounded arithmetic, 322
 - bounded-depth, 235, 319
 - complete, 316
 - conditional lower bound, 318
 - counting, 318
 - depth- d , 319
 - implicationally sound and complete, 316
 - separation from extended, 318
 - simulation of cutting planes, 318
 - simulation of Nullstellensatz, 318
 - simulation of polynomial calculus, 318
 - sound, 316
- fresh variable, 153
- frozen variables, 444, 447, 873
- Fujita, 831
- full scan mode, 1067
- full symmetry breaking, 544
- fully polynomial randomized approximation scheme (FPRAS), 1017
- function
 - abstraction, 804
 - acquisition, 486
 - certifying, 708
 - computable, 15
 - concretization, 804
 - cost, 1092
 - distance, 372, 378
 - effective weight, 1036
 - extension, 1230
 - Herbrand, 1177, 1178, 1205
 - injective, 1257
 - lifting, 300
 - literal weight, 1038
 - partially symmetric, 560
 - pseudo-Boolean, 49
 - ranking, 814
 - recursive, 14
 - residual, 561
 - Skolem, 1178
 - subexponential, 683
 - tail, 1036
 - totally symmetric, 560
 - uninterpreted, 796

- utility, 1092
- function extraction
 - round-based, 1206
- function table, 403
- functional definition, 403, 416, 417
- functional dependency, 403, 416
- functional pigeonhole principle, 256, 275, 277, 321
- functional problems, 616
- functor, 596
- future reductions, 374
- fuzzing, 660

- Gödel's incompleteness theorem, 14
- galloping search, 1028
- Galois correspondences, 597
- Galois field, 415
- game, 1178
 - prover-delayer, 1211
 - two-player, 1211
- game theory, 500
- gate, 1048
 - ancestor, 1049
 - assigned, 1049
 - cardinality, 1056
 - child, 1049
 - descendant, 1049
 - input, 1049
 - justified, 1050
 - output, 1049
 - parent, 1049
 - primary input, 1049
 - primary output, 1049
 - unassigned, 1049
- gate extraction, 417
- Gauss-Jordan elimination, 1030, 1034
- Gaussian
 - elimination, 251, 275, 281, 415, 416, 644
 - reasoning, *see also* Gaussian elimination, 251
- Gelernter, 19
- general
 - cutting planes, 286, 304
 - division, 286
 - SAT, 670
- generalizations, 705
- generalized arc consistency (GAC), 1120
- generalized axiom, 1190
- generalized resolution, 285, 287, 308
- Generalized Unit Clause (GUC), 883
- generation
 - almost uniform, 1019
 - conflict set, 1286
 - model, 1286
- generator, *see also* group, generator, 519
 - almost uniform, 1019
 - irredundant, 519
 - Mace-style, 821
 - redundant, 519
- genetic algorithm, 485
- geometric distribution, 1034
- geometric mean, 360
- George Boole, 10
- George Cantor, 12
- GGA, 485–488
- GGA++, 485, 488
- Gilmore, 19
- global
 - subsumption, 399
 - symmetry, 558
 - variable, 750
- globally blocked clause, 411, 413, 641
- glucose level, 409
- glue, 409
- GlueMiniSat, 402
- Gnovelty+PCL, 486
- goal state, 1346
- good, 1166
- Gottlob Frege, 12
- Gröbner basis, 235, 418
 - computation, 235, 278, 282
 - proof system, *see* polynomial calculus
- Gröbner basis algorithm, 50
- graph, 526, 699
 - acyclic, 713
 - adaptec, 530
 - automorphism, 565
 - bigblue, 530
 - bipartite literal-clause, 617
 - bipartite variable-clause, 617

- colouring, 79, *see also* colouring, 277, 282
- community, 723
- conflict, 244, 617, 700
- connectivity, 107
- consensus, 700
- dual, 700
- expander, *see also* expander graph, 257, 279, 320
- factor, 1360
- implication, 119, 138, 403, 1055, 1106, 1187
- incidence, 699
 - directed, 700
- k -colourable, 277
- obstruction, 711
- pigeonhole principle, 277
- primal, 699
 - positive, 706
- random, *see* random graph
- representation, 1357
- solving algorithms, 300
- tautology formula, *see* ordering principle formula
- torso, 722
- greater-than-or-equal predicate, 543
- greatest common divisor (GCD), 285
- greedy search, 215
- Grigori Tseitin, 22
- ground formula, 1269
- ground resolution, 20
- ground term, 1269
- group, 516, 526
 - action, 523, 524
 - automorphism, 526
 - cyclic, 519
 - detection, 967
 - generators, 519
 - irredundant, 519
 - redundant, 519
 - isomorphic, 517
 - isomorphism, 517
 - MaxSAT, 934, 967
 - problem, 934
 - of negations \mathcal{N}_n , 518
 - of negations and permutations, 522
 - order, 517
 - permutation, 520
 - product
 - direct, 560
 - semi-direct, 523
 - subgroup, 518
 - cyclic, 519
 - proper, 518
 - trivial, 518
 - symmetric, 520
- group-induced equivalence partition, 524
- group-oriented minimal unsatisfiable subformula, 1026
- GSAT, *see* local search, GSAT
- h-modularity, 723, 724
- halting problem, 233
- Hamming
 - ball, 677
 - distance, 677
- hard clause, 905
- hardness, 39, 465
- hardness escalation, 301
- hardware verification, 500, 1159
- harmonic mean, 360
- hash consing, 416
- hash family, 1025
- hash functions, 1035
 - dependent, 1027
 - family, 1020
 - prefix family, 1027
 - probabilistic, 1020
 - row-echelon, 1035
 - sparse, 1026
 - strongly 2-universal, 1025
 - Toeplitz, 1025
 - uniform family, 1025
 - universal, 1016
 - XOR-based, 1025
- heavy-tailed behavior, 463, 465, 466
- heavy-tailed distribution, 126, 465, 466
- HeerHugo, 111, 203
- Henry Kautz, 23
- Herbrand, 15
- Herbrand function, 1177, 1178, 1205

- Herbrand model, 15
- Herbrand's theorem, 15
- heterogeneous base classes, 709
- heteroscedastic noise, 486
- heuristic, 192, 205, 207, *see also* branching heuristics, 351, 378, 1161, 1167
 - adaptive, 378
 - Böhm, 24
 - backbone search, 193
 - branching ordering, 372, 380
 - CDCL, 247, 254, 268
 - clause reduction, 190, 193
 - decision, 186
 - deterministic, 475
 - difference, 186, 192
 - direction, 192, 195
 - distance, 186
 - DLIS, 24
 - double look-ahead, 206
 - Franco, 380
 - Freeman, 24
 - greedy, 24
 - Jeroslow-Wang, 24, 380
 - Johnson, 380
 - MaxSAT, 912
 - optimizations, 999
 - phase selection, 116
 - pre-selection, 192, 199
 - pure-literal-rule, 24
 - random problems, 452
 - search, 883
 - shortest-clause-rule, 24
 - splitting, 116
 - unit-clause-rule, 24
 - value ordering, 384
 - variable ordering, 116
 - variables ordering, 384
 - VSIDS, 24
 - weighted binaries, 193
 - weighted binaries, 189
- heuristic algorithms, 481
- heuristic-equivalence parameter, 476
- hidden literal, 393, 408
- hidden literal elimination (HLE), 408
- hidden structure, 693
- hidden tautology, 408
- hidden tautology elimination (HTE), 408
- hierarchical encodings, 80
- hierarchical hardness model, 491
- hierarchical symmetry, 562
- higher-order symmetry, 562
- Hilary Putnam, 19
- Hilbert's Nullstellensatz, 271
- hints, 635, 823
- hitting community (h-community), 723
- hitting community structure (h-structure), 723
- hitting formula, 577, 711
- hitting set, 698
 - in MaxSAT, 962
 - IP formulation, 962
 - minimum cost, 962
- HLE, 408
- hole, 828
 - set, 828
- holey Latin square, 828
- homogeneity, 355
- homomorphism, 575, 576, 596
 - clause-preserving, 577
- Horn, 28, 705, 1138
 - extended, 28
 - renamable, 28, 707
- Horn formula, 264, 672
 - quantified extended, 1148
- Horn-3-CNF, 396
- house-sitting principle, 272
- HTE, 408
- hybrid strategy, 1008
- hybrid systems, 756
- Hydra, 487, 497, 498, 500
- hyper binary resolution, 393, 396, 405–407, 417, 644
- hyper binary resolvent, 406
- hyper resolution, 127, 405
- hypercube
 - Boolean, 304
- hyperedge, 700
- hypergraph, 612, 699, 700
 - colouring, 612, 613
 - dual, 700
 - square, 614
- hypergraph-colouring

- strong, 626
- weak, 626
- hypertree-width, 721
- ideal, 271
- ideal proof system (IPS), 311
 - noncommutative, 311
- identity
 - quasigroup, 829
 - short conjugate-orthogonal, 829
- identity element, 516
- if-then-else, 417
- image, 807
- immediate-decision oracle, 372
- implementations, 399
- implicant, 514
 - prime, 19, 514
 - essential, 19
- implicate, 514
 - prime, 514, 1139
 - formula, 1139
- implication, 514
 - direct, 202
 - indirect, 202
 - logical, 102, 237
 - unit, 119
- implication graph, 119, 138, 403, 1055, 1106, 1187
 - binary, 403–408
- implication proof, 655
- implication tree, 208
- implication via unit propagation, 1201
- implicationally complete, 293, 308
- implicit BCP, 1000
- implicit hitting set (IHS), 962
- implicit hitting set approach, 962
- importance sampling, 1005
- improper subgroup, 518
- improvements, 769
- in-processing, 1031, 1032
- inactive
 - clause, 208, 209
 - literal, 1332
- incidence graph, 699
 - directed, 700
- incidence treewidth, 713
- incomparable, 275
- domination, 702
- incompatible values, 1290
- incomplete
 - assignment, 1244
 - autarky search, 603
 - Latin square, 828
 - orthogonal array, 837
- inconsistent formula, 1133
- incremental determinization, 1204
- incremental SAT, 153, 392, 403, 412, 413, 943
 - solving, 749
 - in MaxSAT, 942
- incremental solving, 1028
- incrementality, 1286
- incumbent, 484, 485
- independence
 - variable, 1182
 - actual, 1182
- independent component, 400
- independent set, 697
- independent set formula, 258
- index
 - dispersion, 1024, 1025
 - of stability, 466
 - satisfiability, 29
 - selector, 539
- indexing
 - function, 300
 - gadget, 302
- indicator variable, 153
- indirect implications, 202
- induction, 749
 - k -induction, 749
- induction principle, 272
- inductive invariant, 748, 749, 755
- industrial
 - circuits, 1079
- inequality
 - Chebyshev, 1023
 - linear, 156
 - Markov, 1023
- inference rule, 109
- initial sequent, 317
- initial state, 1346
- injective function, 1257
- innermost quantifier block, 1178

- innermost variable, 1169
- inprocessing, 251, 392–394, 398, 400, 409, 416, 418, 637
- input gate, 417
- input refutation, 241
- input resolution proof, 646
- input size, 694
- input variables, 955
- instance
 - exceptionally hard, 465
 - problems, 767
- instance features, 489, 491, 497
- Instance-Specific Algorithm Configuration (ISAC), 492, 496
- instantiation, 1207
- integer linear programming, 282, 288, 293, 905, 1087
- integer programs
 - reduced cost fixing, 965
- integer variables, 1275
- intelligent backtracking
 - modal logics, 1247
- intensional interpretation, 9
- inter-reducible, 1019
- interface equalities, 1286, 1305, 1307, 1308
 - deduction, 1286
- interior point algorithms, 45
- internal node, 113, 197, 646, 1337
- internal variable, 412
- interpolant, 411, 635, 750
 - generation, 1031
 - preliminary, 750
 - SMT, 1314
- interpolating quadruples, 750
- interpolation, 305, 750, 753, 814
 - feasible, 1210
 - method, 259, 321
- interpretation, 4, 1270
 - extensional, 9
 - intensional, 9
- intersection, 516
- intractability
 - fixed-parameter, 697
- invariant strengthening, 755
- invariants, 769
- inverse-method approach, 1224
- IR+calc, 1208
- IR-calc, 1207
- iRace, 488
- irrational magic square, 844
- irredundant clause, 398
- irredundant CNF, 403
- irredundant generator, 519
- irredundant sum-of-products, 403
- irregularity, 22, 658
- ISAC, 492–494, 496
- isolation procedure, 686
- isomorphic, 698
 - CNF formula, 698
- isomorphic group, 517
- isomorphism elimination, 845
 - single clause, 849
- isotopic quasigroups, 849
- iterated local search, 484, 486
- iterative racing, 488

- J. Zhang, 849
- Jaccard distance, 493
- Jan Łukasiewicz, 16
- Jeroslow-Wang heuristics, 380
- John Venn, 11
- Johnson estimator, 380, 381
- Johnson heuristic, 380–382
- jump depth, 198
- justification frontier, 1063

- k -CNF formula, 236
- k -DNF resolution, 319
 - size space trade-off, 319
- Karp-Luby counter, 1033
- $KBKF_n$, 1210
- kernel, 696
 - lean, 589
 - problem, 696
- kernelization, 393, 696
- Kirkman triple system, 840
- Kissat, 409
- Kleene, 15
- Klein 4-group V , 517
- KLM counter, 1034
- knapsack 0-1 problem, 1114
- knowledge compilation, 1001
- Kripke structure, 741, 1226
- Kurt Gödel, 14

- kurtosis, 466
- label, 967, 1228
- labeled CNF framework, 967
- labeled formula, 1228, 1257
- labelled clause-set, 611
- ladder encoding, 81
 - relaxed, 81
- Lagrange, theorem of, 519
- Lagrangian method, *see* discrete Lagrangian method
- lambda expressions, 1275
- landscape, 214
- language, 17, 237
- large sets, 831
- Lasserre proof system, 311
- last literal, 539
- Latin rectangle, 823
- Latin square, 821
 - conjugate, 826
 - embedded, 823
 - frame, 828
 - holey, 828
 - idempotent, 824
 - incomplete, 828
 - orthogonality, 824
 - partial, 823
 - reduced, 849
 - self-orthogonal, 826
- lattice, 516
- layered theory solvers, 1302
- layers, 1255
- lazy
 - abstraction, 808, 811
 - approach, 1285, 1310
 - approach to SMT, 1269, 1285, 1308, 1310
 - abstract, 1293, 1312
 - offline, 1286, 1297
 - online, 1287, 1297
 - splitting on demand, 1300
 - data structures, 149, 1104, 1356
 - SMT solvers, 1285
- LBD, 409
- LDPC, 1026
- lean
 - approaches, 1255
 - clause-sets, 589
 - kernel, 589
 - lean kernel problem, 616
 - lean type, 1255
- LeapsAndBounds, 489
- learning, 355, 1167, 1243, 1358
 - clause, 23, 25, 121, 140, 243, 269, 392, 394, 398, 399, 409, 472, 477, 1055, 1166
 - conflict driven, 1055
 - good, 1164, 1166
 - local, 202, 626
 - nogood, 1164
 - parity, 90, 93
 - recursive, 417
 - reinforcement, 498
 - SMT, 1288, 1291, 1294, 1296, 1301
 - static learning in SMT, 1290
- learning-rate based branching (LRB), 270
- least fixpoint semantics, 746
- least number heuristic, 849
- least number principle, *see* ordering principle formula
- legal k -colouring, *see* colouring
- length, 308
 - clause, 669
 - cutting planes, 302
 - cycle, 521
 - lifting, 300
 - maximum, 702
 - polynomial calculus, 274
 - resolution, 239, 255
 - v.s. width, 260, 261
- length-space trade-off, 266, 268
- length-width
 - lower bound, 259, 260
 - trade-off, 267
- Leon Henkin, 17
- leptokurtic distribution, 466
- leq predicate, 515
- less-than-or-equal predicate, *see* leq predicate
- level, 1178
 - assertion, 121, 246
 - decision, 137, 243, 1167

- of a literals, 1162
 - of a variable, 1161
- lex-leader, 563
- lex-leader predicate, 537
- lifted CNF formula, 301
- lifting, 300, 303
 - auxiliary clause, 301
 - CNF formula, 301
 - functions, 300
 - indexing gadget, 302
 - length, 300
 - lifted clause, 301
 - main clause, 301
 - main variable, 301
 - original clause, 301
 - relation, 300
 - search problem, 300
 - selector variable, 301
 - XOR-substitution, 302
- linear
 - encoding, 745
 - equation over the two-element field, 672
- linear autarky, 24, 29, 604, 609, 610
- linear combination, 273
- linear inequalities, 156
- linear program relaxation, 299, 310
- linear programming, 45
 - mixed integer, 1363
- linear programming relaxation (LPR), 1115, 1117
- linear pseudo-Boolean problem, 82
- linear temporal logic, 740
- Lingeling, 403, 410, 412, 415, 487
- literal, 236, 564, 637, 669, 698, 766, 931, 1015, 1146, 1178, 1332
 - 1UIP, 244
 - active, 1332
 - asserting, 244
 - asymmetric, 407–409
 - axiom, 284
 - black and white, 682
 - blocked, 1200
 - Boolean, 512
 - conjunction, 1304
 - covered, 410
 - decision, 242
 - deduction, 1288
 - degree, 617
 - empty, 1332
 - encoding, 653
 - equivalent, 393, 396, 404, 406, 416, 417
 - existential, 1161, 1332
 - failed, 187, 188, 395–397, 405, 406, 408, 411
 - fixed, 411
 - hidden, 408
 - inactive, 1332
 - last, 539
 - level, 1162
 - merged, 1181
 - monotone, 588, 1162
 - negative, 637, 1332
 - ordering, 1146
 - positive, 28, 637, 1332
 - pure, 20, 38, 204, 314, 394, 396, 403, 445, 588, 640, 699, 1162, 1167, 1339
 - elimination, 699, 702
 - randomized, 1332
 - sorting, 652, 653
 - unit, 1186
 - weight functions, 1038
 - witness, 413
- literal block distance (LBD), 148, 270
- literal weight functions, 1038
- literal-regular clause-set, 617
- liveness, 741, 747
- LK, 316
 - proof, 317
 - refutation system, 318
 - rules of inference, 317
- local
 - cluster, 185
 - learning, 202, 626
 - minimization, 145
 - symmetry, 558
 - variable, 750
- local search, 42, 90, 213, 484, 485, 491, 492, 497, 603, 1358
 - AdaptNovelty**, 214
 - HSAT**, 219
 - R+AdaptNovelty+**, 214

- TSAT, 219
- UBCSAT, 214
- UnitWalk, 218
- Walksat, 216
- adaptg2wsat+, 214
- gNovelty+, 214
- algorithms, 878
- clause weighting, 218
- DLM, 219
- flooding, 218
- GSAT, 42, 215
- iterated , 484, 486
- Novelty, 43
- PAWS, 219
- RSAPS, 219
- SAPS, 214, 219
- stochastic, 213, 1358
- WalkSAT, 42
- weights, 42, 43, 218
- log encoding, 79, 80, 92, 93
- log support encoding, 80
- logarithmic width, 305
- logic
 - categorical, 6
 - difference, 1273, 1281, 1284, 1285, 1292, 1293, 1307, 1315
 - first-order, 399, 419, 1215
 - four-valued, 1073
 - propositional, 155
 - quantification, 12
 - sylogistic, 6
 - categorical propositions, 6
- logic machine, 10
- logic Piano, 11
- logical
 - axiom, 317
 - conjunction, 512
 - disjunction, 512
 - implication, 237, 316
 - matrix, 16
 - negation, 512
- logical truth, 5
- logically equivalent, 397, 417, 1131
 - formula, 638, 1133
- logicism, 12
- long-distance Q-resolution, 1180, 1181, 1206, 1210
 - derivations, 1189
 - function extraction, 1205
 - long-distance resolution, 1180
 - look-ahead, 186, 372, 395, 396, 416, 644, 1000
 - autarky resolvents, 205
 - breadth-first, 24
 - depth-first, 24
 - discussion, 372
 - double, 202, 205, 206
 - heuristics, 206
 - resolvents, 207
 - reduction, 372
 - strategy, 183
 - tree-based, 208, 209
 - look-ahead architecture, 185
 - look-ahead autarky clause, 625
 - loop detection, 814
 - loop unwinding, 800
 - Lovász-Schrijver proof system, 311
 - low resolution, 968
 - lower bound, 963, 998, 1006, 1113, 1115
 - average-case, 258, 259, 278
 - length-width, 259
 - limit, 1115
 - linear programming relaxation, 1115
 - maximum independent set, 1114
 - MaxSAT, 906
 - underestimation, 907
 - lower-bound technique, 1210
 - LSQG, 831
 - LTL, 740, 745, 798
 - Luby
 - restarts, 148, 269
 - sequence, 269
 - Ludwig Wittgenstein, 16
 - Ma, 831
 - MACE, 831
 - Mace-style model generator, 821
 - magic number, 843
 - magic square, 843
 - diagonally ordered, 844
 - elementary, 844
 - irrational, 844

- nonelementary, 844
- normal, 843
- rational, 844
- symmetric, 844
- main variable, 300
- majority rule, 24
- majority voting, 491
- MAJSAT, 1334
- MAP, 1037
- MapleSAT, 402
- March, 415, 416
- marginal
 - contribution, 495
 - formula, 578
 - probability, 382, 1002, 1009
- Markov Chain Monte Carlo (MCMC),
 - 996, 1003, 1004
- Markov inequality, 1023
- Martin Davis, 19
- matching, 724
- matching autarky, 608
- matching principle, *see also* parity principle, 272
- matching satisfiable clause-set, 608
- matrix, 51, 583, 1134, 1331
 - basic, 574
 - clause-variable, 611
 - logical, 16
 - random, 1035
 - sign-central, 619
 - minimal, 619
 - standardised, 611
 - strongly symmetric, 845
 - variable-clause, 611
 - witness, 623
- MAX-2-SAT, 44
- Max-2-SAT, 703, 704
- MAX-SAT, 44, 214
- Max-SAT, 703
- maximal
 - autarky, 593
 - deficiency, 572
 - formula, 578
 - literal-degree, 617
 - variable-degree, 617
- maximality clauses, 80
- maximum cardinality search (MCS),
 - 109
- maximum deficiency, 725
- maximum expected utility (MEU),
 - 1334
- maximum independent set, 1114
- maximum length, 702
- maximum satisfiability, 43, 134, 158,
 - 214, 929, 1123
- MaxSAT, 323, 419, 481, 485, 904,
 - 929, 931, 1037
- F^b transformation, 940
- algorithms, 939
 - core-guided, 930, 948
 - Fu and Malik, 948
 - implicit hitting set, 930, 962
 - incomplete, 968
 - Integer Programming, 966
 - linear SAT-UNSAT, 946
 - Maxino, 959
 - model-improving, 930, 946
 - modern, 939
 - MSU3, 952
 - OLL, 954
 - parallel, 969
 - PM2, 953
 - PMRes, 958
 - portfolios, 969
 - stochastic local search, 968
 - WPM1/WMSU1, 950
 - WPM3, 960
- applications, 936
 - AI, 937
 - bioinformatics, 938
 - combinatorial problems, 937
 - configuration, 936
 - CSP, 938
 - MCS, 938
 - planning, 936
 - propagation complete encodings, 938
 - scheduling, 936
 - security, 937
 - verification, 937
- blocking variables, 940
- Boolean multilevel optimization,
 - 933
- cardinality constraints, 944

- core, 932
- cost function, 929
- decision problem, 944
- decision problem encoding, 946
- dual-rail, 323
- equivalence, 941
- equivalent instance, 904
- evaluations, 485, 917
- feasible solution, 932
- formalism, 918, 931, 939
- formula
 - partial, 932
 - unweighted, 932
- group, 934, 967
 - detection, 967
- hard clauses, 931
- heuristics, 912
- inference rule, 909
- Max-*k*SAT, 904
- MaxSAT evaluations, 930
- MaxSAT resolution, 942
- optimal solution, 932
- partial, 905, 932
- partitioning, 971
- preprocessing, 966
 - binary core removal, 967
 - label matching, 967
 - SAT-based, 966
 - subsumed label elimination, 968
- preprocessor
 - Coprocessor, 968
 - MaxPre, 968
- problem definition, 932
- relaxation variables, 941
- sequence of SAT approaches, 943
- soft clause cloning, 942
- soft clauses, 931
- solvers
 - Eva, 959
 - LinSBPS, 968
 - LMHS, 962
 - MaxHS, 962
 - Maxino, 959
 - Open-WBO, 952
 - Open-WBO-Inc, 969
 - Pacose, 947
 - PWBO, 970
 - QMaxSAT, 947
 - RC2, 957
 - using assumptions, 942
 - using incremental SAT, 942
 - weighted, 905, 932
 - weighted clauses, 931
- maxterm, 513
- McCluskey, 19
- McCune, 843
- mean, 1022
 - arithmetic, 360
 - geometric, 360
 - harmonic, 360
 - power, 360
- measure, 366
 - for upper bounds, 373
 - on trees, 365
- measure-and-conquer, 373
- mechanical process, 9
- median, 1022
- medieval period, 8
- memoization, 1344
- merged literal, 1181
- merging rule, 1098
- message passing
 - algorithms, 889, 890
 - belief propagation, 890
 - survey propagation, 891
 - warning propagation, 890
- message-damping variant, 1009
- meta-algorithm, 482, 489
- meta-algorithmic techniques, 481, 496, 499
- meta-approach, 969
- method
 - basic local search, 218
 - cavity, 224, 873
 - cyclic zipper, 834
 - discrete Lagrangian, 219
 - interpolation, 259
 - Nelson-Oppen, 1303, 1304, 1312
 - replica, 868
- MGTP, 831
- microstructure, 565
- mini-scoping, 1183
- minimal
 - autarky, 594

- encoding, 824
- literal-degree, 617
- true subtree, 1166
- unsatisfiability, 571, 578–581, 583
 - problem, 615, 622
 - variable-degree, 617
- minimal false formula, 583
- minimal false subtree, 1164
- minimal falsity, 583
- minimal monotonic structures, 41
- minimal unsatisfiability
 - generalised, 614
- minimal unsatisfiable, 571, 725
 - conjunctive normal forms, 571
 - core, 635
 - expression, 30
 - formulas, 571, 572, 579, 725
 - marginal, 578
 - maximal, 578
 - splitting, 573
 - Horn formulas, 575
- MINIMAX trees, 1339
- MiniMaxSat, 913
- minimization
 - clauses, 255, 399
 - local, 145
 - recursive, 145
- MiniSAT, 654
- MiniSat, 163, 392, 399, 407
- Minisat, 489
- miniscoping, 1183
- minterm, 513
- MIS, 1026
- mixed integer linear programming (MIP),
 - 299, 306, 310
- mixed integer programming, 492, 497
- mixed symmetry, 526
- mixed-integer cut, 1118
- mobility, 219
- modal
 - backjumping, 1247
 - conflict set, 1247
 - DPLL, 1231, 1238, 1239, 1242
- modal atom, 1231
- modal logics, 1225
 - non-normal, 1228
 - normal, 1225, 1227
- modal operators, 1223
- model, 33, 136, 411, 584, 699, 792,
 - 1015, 1270, 1286, 1313
 - abstract, 804
 - empirical performance, 485, 491
 - equivalence, 1144
 - fault, 1065, 1066
 - generation, 1286
 - Herbrand, 15
 - refined, 813
 - satisfiability, 1142
 - tree search, 468
- model checking, 391, 411, 739, 1159
- model counter
 - ApproxCount, 1003
 - BPCount, 1008
 - CDP, 997
 - MBound, 1007
 - MiniCount, 1009
 - Relsat, 998
 - SampleCount, 1006
 - SampleMinisat, 1005
 - c2d, 1001
 - XOR streamlining, 1007
- model counting, 160, 400, 416, 712,
 - 716, 993, 1015, 1039
 - approximate, 1003, 1015, 1018
 - computational complexity, 994
 - deterministic, 1018
 - exact, 996
 - FPRAS, 996
 - hashing-based, 1016, 1020, 1039
 - probabilistic, 1018, 1020
 - problem, 1015
 - Stockmeyer’s technique, 1035, 1037,
 - 1038
 - weighted, 1035
- model generation
 - finite domains, 819
 - tools, 819
- model theory, 4, 15
- model-improving algorithms, 946
- model-improving approximation, 968
- modular arithmetic, 791
- modular treewidth, 720, 721
- modularity, 723
- monadic second-order, 716

- monoid
 - autarky, 591, 593, 596
 - of partial assignments, 591
- monomial, 270
 - space, 274
- monotone
 - circuit, 305
 - literal, 588, 1162
 - property, 39
 - real circuit, 321
- monotone-input reduction, 409
- Montague-Scott structure, 1229
- Monte Carlo, 1016, 1017, 1032
- most probable explanation (MPE), 937, 1334
- move
 - freebie, 216
 - greedy, 216
 - random walk, 216
 - sideways, 215
 - uphill, 216
 - winning, 1193
- MPE, 1039
- MSO, *see* monadic second-order
- multi-games, 1198
- multi-linear polynomials, 49
- multilinear, 270, 274
- multilinearization, 273
- multiplexer, 417
- multiplication, *see also* cutting planes, *see also* polynomial calculus, 273, 284
- multiplication rule, 1096
- multiplicative approximation amplification, 1024
- multistart random walks, 674, 675
- multivalued encoding, 80
- mutexes, 769
- mutually consistent subtrees, 1240
- mutually-inconsistent subtrees, 1241
- MUX, 417
- myopic algorithm, 37

- N-queens, 83
- nauty, 563, 565
- necessary assignment, 203
- necessity, 5

- negation
 - logical, 512
 - normal form, 580
 - rule, 1096
- negation axiom, 274
- negation normal form, 742
 - quantified Boolean, 1134
- negative literal, 236, 637, 1332
 - vertices, 532
- Negative Normal Form (NNF), 1225
- Nelson-Oppen method, 1303, 1304, 1312
- nested formula, 30
- nested satisfiability, 30
- NiVER, 402
- NNF, 742
- node
 - internal, 113, 197, 646, 1337
- nogood, 1166, 1343
- noise, 216
- non-automatability, 282
- non-automatable, 251, 282, 305
- non-backtracking algorithms, 452
- non-chronological backtracking, 118, 119, 140, 144, 246, 1055, 1343, 1357
- non-CNF, 76
- non-relaxable clause, 905
- non-shared-input elimination, 409
- non-trivial subgroup, 518
- non-uniform class, 687
- nondeterministic solver, 252
- nonelementary magic square, 844
- nonlogical axiom, 318
- normal autarky system, 606
- normal form, 994
 - conjunctive, 1015
 - disjunctive, 1015, 1032
- normal magic square, 843
- normalization
 - modal logics, 1243
 - SMT, 1290
- normalized form, 283, 286
- notation
 - asymptotic, 238
 - big-O, 238
 - big-O-tilde, 307

- cycle, 520
- NP-complete, 133, 233
- Nullstellensatz, 234, 235, 270, 282, 321
 - degree, 272, 275
 - multilinear, 273
 - refutation, 270
 - size, 271, 275
 - size-degree trade-off, 281
- number
 - covering array, 836
 - even, 1007
 - magic, 843
 - odd, 1007
 - of clauses, 37
 - of solutions, 1005
 - Ramsey, 832
 - random, 476
 - Van der Waerden, 833
- number P, *see* #P
- numerically least solutions, 537

- \mathcal{O}^* -notation, 696
- OBDD proof system, 312
- OBDD-based approach, 1224
- observability don't care, 1060
- observable, 1346
- observation propositions, 1346
- obstruction graph, 711
- occurrence list, 397–400
- odd number, 1007
- offline schema, 1286
- on-line lazy approach, 1224
- on-the-fly subsumption, 399, 402, 403
- one-sided approximation, 1019
- onto functional pigeonhole principle, 256, 275, 277, 321
- operation
 - associative, 516
 - join, 9
 - meet, 9
 - parallel-update, 1276
- operations research, 1362
- operators
 - Boolean, 10
 - modal, 1223
- optimal policy, 474
- optimality of parallel plans, 772
- optimization
 - Boolean multilevel, 933
 - heuristics, 999
 - problems, 703
- optimization problem
 - MaxSAT, 929
- or-subformula, 581
- or-subtree, 581
- oracle, 1017
 - immediate-decision, 372
 - SAT, 1030
- orbit, *see also* group-induced equivalence partition, 524
 - partition, 525
 - pruning, 554
- order, 821
 - branch, 380
 - group, 517
- ordered binary decision diagram (OBDD), 54, 311
- ordered partition, 516, 527
 - pair, 549
 - refinement, 527
- ordering
 - heuristics
 - branching, 372, 380
 - value, 384
 - variables, 116
- ordering principle, 261
- ordering principle formula, 261, 276
- Organon, 6
- original encoding, 744
- orthogonal
 - array, 836, 837
 - incomplete, 837
- orthogonality
 - conjugate, 827
 - Latin square, 824
- outer resolvent, 1201
- outermost quantifier block, 1178
- output variables, 955
- overlap clause, 698

- P+ForallRed, 1214
- PAC, 1018, 1019, 1032
- parallel classes, 840

- parallel plans, 770
 - optimality, 772
- parallel portfolio, 497
- parallel reasoning, 400, 409
- parallel solving
 - MaxSAT, 969
 - portfolios, 970
 - search space splitting
 - MaxSAT, 970
- parallel-update operation, 1276
- parameter, 483, 494, 496, 498, 694, 695
 - conditional, 483
 - dual, 704
 - heuristic-equivalence, 476
 - hybrid satisfiability, 721
 - satisfiability, 693, 694, 701, 721
- parameter control, 482, 498
- parameter importance, 482, 498
- parameter settings, 482
- parameterization, 693
- parameterized
 - complexity, 694
 - reduction, *see* fpt-reduction
 - SAT, 700
- ParamLS, 483, 485–488
- parent clauses, 590
- parenthesis theorem, 408
- Paris-Wilkie translation, 322
- parity
 - constraint, 82, 90, 1007
 - formula, 416, 1210
 - learning, 90, 93
 - principle, 319
 - formula, 321
 - reasoning, 414, 418, 419
- parking, 150
- parsimonious, 995
- parsimonious reduction, 995
- part
 - autarky, 640
 - conditional, 640
- partial
 - assignment, 135, 236, 586, 637, 1207, 1292
 - autarky search, 594
 - monoid, 591
 - safe, 625
 - SMT, 1292
 - expansions, 1193
 - Latin square, 823
 - MaxSAT
 - formula, 932
 - problem, 905
 - SBP, 544
 - truth assignment, 1231
- partially symmetric function, 560
- partition, 515, 526, 833, 1020
 - block, 515
 - cell, 515
 - classes, 515
 - conflict, 1109
 - discrete, 515
 - function, 1009
 - intersection, 516
 - lattice of, 516
 - orbit, 525
 - ordered, 516, *see also* ordered
 - partition, 527
 - refinement, 527
 - reason, 1109
 - refinement, 527, 552
 - union, 516
 - unit, 515
- PASSAT, 1065
- path, 798
- path slicing, 814
- Paul Beame, 23
- Payley-Zygmund inequality, 1023
- PBO, 1092
- PBS, 1091
- PC, *see* polynomial calculus
- PCR, *see also* polynomial calculus
 - resolution, 274
 - negation axiom, 274
- peak of hardness, 999
- pebble game, 263
- pebbling, 263
 - formula, *see* pebbling contradiction
 - space, 263
 - trade-off, 263
- pebbling contradiction, 263, 264, 272, 276, 280, 281

- lifted, 302
- substituted, 264, 268, 280, 281
- XOR-ified, 264
- per-constraint encoding, 1283, 1290
- per-instance algorithm selection, 489
- per-instance algorithm selector, 490
- perfect Mendelsohn design, 842
 - definition, 842
- performance guarantee, 693
- performance model
 - empirical, 485
- performance optimization, 482
- permanent, 995
- permissive predicate, 701
- permutation, 358, 520, 526
 - phase-shift, 539
 - predicate, 537, 539
 - predicate $PP(\pi; X)$, 537
 - search tree, 549, 550
 - set, 549
 - signed, 560
 - single, 555
 - support, 521
- phase, 401
- phase saving, 247, 254, 401
- phase selection
 - heuristic, 116
- phase transition, 37, 93, 152, 222, 438, 446, 454, 861, 869, 870, 872, 999
 - clustering transition, 869
 - phenomenon, 222
- phase-shift cycle, *see also* cycle, phase-shift, 539
 - first, 539
- phase-shift permutation, 539
- phenomenon
 - clustering, 869
 - phase transition, 222
- PHP, *see also* pigeonhole principle, 392
- picosat, 654
- piecewise regular input resolution proof, 646
- pigeonhole formula, 22, 392, 403
- pigeonhole principle, 256, 277, 279, 302, 304, 310, 313, 315, 318–322
 - functional, 277
 - graph, 277
 - onto functional, 256, 275, 277, 321
 - pseudo-Boolean form, 309
 - weak, 256, 319, 321
- pigeonhole principle formula, 262, 272, *see also* pigeonhole principle
- pivot variable, 1179
 - universal, 1181
- PK, 316
- Plaisted-Greenbaum encoding, 417
- plan, 755, 767, 770
 - parallel, 770
 - sequential, 767
- planning, 90, 936, 1158, 1335, 1346, 1348, 1352, 1353
 - classical, 766
 - contingent, 781, 782
 - temporal, 779
 - with full observability in NP, 785
 - with partial observability in NP, 783
- plateau, 215
- player
 - existential, 1178
 - universal, 1178
- Plingeling, 412
- PMD, 842
- pointer, 795
- polarity, 86, 417
- policy, 1143
 - optimal, 474
 - tree, 1355
- polynomial, 270, 418
 - calculus, *see* polynomial calculus
 - degree, 274
 - ideal, 271
 - multi-linear, 49
 - representation
 - unique, 50
 - ring, 270
- polynomial calculus, 234, 235, 272,

- 273, 275, 1214
- Boolean axiom, 273
- characteristic of field, 281
- degree, 274, 275
- incomparable to cutting planes, 303
- length, 274
- linear combination, 273
- monomial space, 279
- multilinear, 274
- multiplication, 273
- practical algebraic calculus (PAC), 282
- proof logging, 282
- refutation, 273
- resolution, *see* polynomial calculus resolution
- separation
 - from cutting planes, 303
 - from resolution, 275
- size, 274, 275
- space, 274, 279, 280
- symmetry, 322
- trade-off, 280
 - size-degree, 281
 - size-space, 281
 - space-degree, 280
- with resolution, *see* polynomial calculus resolution
- polynomial calculus resolution, 274, 275
 - negation axiom, 274
 - simulation of resolution, 275
- polynomial time
 - autarkies, 604
 - uniform, 694
- polynomially bounded, 237
- polynomially simulate, 237
- pool resolution, 254, 255
- portfolio, 160, 475, 481, 489, 497
 - algorithm, 969
 - approaches, 970
 - parallel, 487, 497
- POS, *see* product-of-sums
- posiform, 50
- positive clause, 36
- positive literal, 28, 236, 637, 1332
- vertices, 532
- Positivstellensatz proof system, 311
- possibility, 5
- post-condition
 - strongest, 813
- power
 - mean, 360
 - pruning, 468
- PP, 996
- ppfolio, 498
- PPSZ algorithm, 675
- PR, 410, 413
- practical algebraic calculus (PAC), 282
- Prawitz, 19
- pre-processing, 1031, 1032
- pre-selection heuristics, 192, 199
- pre-solver, 491
- pre-solver schedule, 492, 494, 497
- pre-solving schedule, 490
- precondition, 767
 - effect, 767
- predicate
 - bit, 540
 - Boolean, 537
 - greater-than-or-equal, 543
 - less-than-or-equal, 515
 - lex-leader, 537
 - partitioning, 808
 - permissive, 701
 - permutation, 537, 539
 - symmetry-breaking, 511
- predicate abstraction, 792, 805
- predicate partitioning, 808
- prefix, 583, 1134, 1160, 1331
 - type, 1137
- prefix hash family, 1027
- prefix-family, 1028
- prefix-slice, 1028
- preimage computations, 753
- preliminary interpolant, 750
- prenex conjunctive normal form, 1178
- prenex form, 1159
- prenex normal form, 1134
- preprocessing, 251, 255, 391, 637, 966, 1051, 1171, 1198, 1203
- CNF, 409

- in SAT solving, 391
 - MaxSAT, 966
 - structure-based, 414
- Presburger arithmetic, 1273
- primal graph, 699
 - positive, 706
- primal treewidth, 713
- primary input, 417
- prime implicant, 19, 514
- prime implicate, 514, 1139
 - formula, 1139
- principal variable, 674
- Principia Mathematica, 13
- principle
 - autarky, 603
 - black-and-white, 603
- priority queue, 402
- probabilistic
 - algorithms, 26
 - complexity classes, 1017
 - polynomial time, 996, 1334
- probabilistic additive approximate model
 - counter, 1018
- probability
 - marginal, 382, 1002, 1009
 - threshold, 1354
- probability amplification, 1022
- probability distribution
 - adorning trees, 363
 - branching tuples, 362
 - tree, 363
- probably approximately correct (PAC), 1018
- probe order backtracking, 36
- probing, 393, 395, 408, 409, 411
 - failed-literal, 395
- problem
 - AI, 937
 - algorithm
 - configuration, 482
 - selection, 489
 - autarky
 - existence, 599, 615, 622
 - non-trivial, 616
 - quasi-maximal, 616
 - Boolean satisfiability, 562
 - combinatorial, 695, 937
 - combinatorial design, 821
 - complexity, 354
 - absolute, 371
 - relative, 371
 - counting, 994
 - data analysis, 937
 - decision, 615
 - functional, 616
 - group MaxSAT, 934
 - halting, 233
 - kernel, 696
 - knapsack 0-1, 1114
 - lean kernel, 616
 - minimal unsatisfiability, 615, 622
 - model counting, 1015
 - optimization, 703
 - promise, 615
 - satisfiability, 615
 - social golfer, 841
 - strict, 701
 - verification, 701
 - vertex cover, 695
- problem kernel, 696
- ProbSAT, 487
- procedure
 - Davis-Putnam, 401
 - Davis-Putnam-Logemann-Loveland (DPLL), 887, 1236
 - classic, 1238, 1240
 - isolation, 686
 - concentrating, 686
 - sparsification, 684
 - Tseitin, 581
- process
 - effective, 9, 14
 - mechanical, 9
- product, 513
 - rule, 353
 - semi-direct, 523
- product term, *see* implicate
- product-of-maxterm representation, 513
- product-of-sums, *see also* conjunctive normal form, 514
- production
 - proof, 653
 - parallel, 655

- program, 17
 - Boolean, 805, 806
- program analysis, 795
- program counter, 792
- program location, 792
- Programming by Optimization (PbO), 483, 494
- projection, 372
 - canonical, 357
 - evaluation, 357
- projection functions
 - alternative, 369
 - axioms, 368
- promise problem, 615
- proof, 237
 - applications, 659
 - as refutation, 238
 - certification, 155
 - checking, 658
 - backward, 658
 - resolution, 657
 - clausal, 635
 - complexity, *see* proof complexity
 - compression, 651, 658
 - LowerUnits, 658
 - RecyclePivots, 658
 - DAG, 239
 - DRAT, 650
 - DRUP, 650
 - format, 128, 645
 - Frege, 316, 317
 - hint, 654
 - implication, 655
 - input resolution, 646
 - logging, *see* proof logging
 - of unsatisfiability, 238, 1202
 - production, 653
 - parallel, 655
 - re-encoding, 655
 - regular, 646
 - resolution, 127, 635, 653, 654
 - satisfiability, 1202
 - SMT, 1313
 - system, *see also* proof system, 640
 - cutting plane, 1096
 - tautology, 655
 - trace, 127, 249, 647
 - trivial, 646
 - with hints, 654, 657
 - without hints, 654, 658
 - witness, 650
- proof checking, 155
- proof complexity, 121, 234, 237, 442
 - propositional, 726
 - random problems, 443
- proof logging, 235, 249, 299, 312
 - DRAT, 299
 - pseudo-Boolean, 299
- proof search, 644
- proof system, 237, 238, 392, 639, 1178
 - algebraic, 270, 311
 - bounded-depth Frege, *see* bounded-depth Frege
 - bounded-depth Frege with parity, *see* bounded-depth Frege, with parity
 - bounded-depth LK with parity, 321
 - completeness, 237, 240
 - cutting planes, *see* cutting planes
 - DRAT, *see also* DRAT, 313
 - extended Frege system, *see* extended Frege system
 - extended resolution, *see* extended resolution
 - Frege system, *see also* Frege system, 316
 - ideal proof system (IPS), 311
 - interference-based, 1201
 - k -DNF resolution, 319
 - Lasserre, 311
 - Lovász-Schrijver, 311
 - lower bound, 1210
 - noncommutative ideal proof system, 311
 - Nullstellensatz, *see* Nullstellensatz
 - OBDD, 312

- polynomial calculus, *see* polynomial calculus
- pool resolution, *see* pool resolution
- Positivstellensatz, 311
- propositional, 1179
- QBF, 1179
- RegWRTI, *see* RegWRTI
- resolution, *see also* resolution, 638
- semialgebraic, 271, 311
- separation, 1210
- sequent calculus, 316
- Sherali-Adams, 311
- simulation, 1209
- soundness, 237, 240
- stabbing planes, 306
 - relation to cutting planes, 306
- strong, 641
- Sum-of-Squares, 311
- proof theory, 3
- proof tracing, 154
- prop_z heuristic, 199
- propagate, *see also* unit propagation, 288
- propagation, 394, 408, 550
 - belief, 225, 454, 890, 1008
 - implication via unit, 1201
 - redundancy, 642, 643
 - reverse unit, 408, 638
 - survey, 224, 454, 1359
 - unit, 115, 136, 185, 242, 393, 394, 637, 699, 702, 1103, 1136, 1161, 1338, 1339
 - BCP, 208
 - warning, 893
- propagation-redundant clause, 410, 411, 413
- proper subgroup, 518
- property, 39
 - assertive, 244
 - extremal, 269
 - monotone, 39
 - reachability, 798
 - safety, 804
- propositional
 - atom, 1231
 - encodings, 743
 - logic, 155
 - proof systems, 1179
 - skeleton, 797
 - variables, 135, 512, 832, 904, 931
 - propositional abstraction, 1272
 - propositional formula, 316
 - cohesion, 580
 - propositional satisfiability problem, 693
 - propositional skeleton, 797
 - propositional temporal logic (PTL), 740
 - propositionally
 - satisfiable, 1233
 - unsatisfiable, 1272
 - valid, 1233
 - propositions
 - categorical, 6
 - observation, 1346
 - random, 1346
 - prover-delayer game, 1211
 - pruning, 1338, 1339
 - coset, 554
 - orbit, 554
 - sibling, 1000
 - pruning power, 468
 - pseudo-Boolean, 418, 1038
 - basic algorithm, 50
 - cardinality constraint reduction, 296
 - competition, 300
 - conflict analysis, 288, 290, 293, 295, 298, 308
 - using division, 293
 - using saturation, 290
 - conflicting constraint, 288
 - constraint, 282, 944
 - cost function, 1092
 - DDT algorithm, 50
 - decision problem, 1091
 - forms, 49
 - formula, 282
 - function, 49
 - fusion resolution, 297
 - integer arithmetic, 293
 - lazy data structures, 1104
 - linear, 50

- linear constraint, 1088
- linear program relaxation, 299, 310
- linearization, 1091
- non-linear constraint, 1089
- normalization, 1090
- normalized form, 283
- optimization, 287, 935
- optimization problem, 1092
- optimum, 1092
- posiform, 50
- proof logging, 299
- propagating constraint, 288
- propagation, 298
- quadratic, 50
- quadratic optimization, 1093
- reason constraint, 290, 291
- reduced reason constraint, 292
- reduction algorithm, 291, 294, 295
- resolved constraints, 285
- slack, 288
- solver, *see* pseudo-Boolean, solving
- solver competition, 300
- solving, 235, 282, 287, 309
 - cardinality constraint detection, 310
 - CNF, 293, 299
 - collapse to resolution, 293
 - comparison to CDCL, 292, 298
 - conversion to CNF, 287
 - exploiting power of cutting planes, 310
 - sensitivity to encoding, 293, 299
 - strengthening rule, 296
 - translation to MaxSAT, 935
 - translation to SAT, 1118
 - utility function, 1092
 - watched literals, 1104
 - weakening, 290
- pseudo-Boolean constraints, 157
 - encoding, 157
- PSPACE, 740, 756, 996
- PSPACE-complete, 268
- pure, 1304
 - literal, 314, 1339
 - elimination, 699, 702
 - literal rule, 313, 314
 - pure additive weighting scheme (PAWS), 219
 - pure autarky, 607, 608
 - pure literal, 20, 38, 204, 394, 396, 403, 445, 588, 699, 1162, 1167, 1339
 - detection, 1136
 - elimination, 699, 707
 - existential, 1199
 - modal logics, 1239
 - quantified Boolean, 1136
 - random problems, 453
 - universal, 1200
 - pure literal filtering
 - modal logics, 1248
 - SMT, 1293
 - pure variable elimination, 1338
 - pure-literal-rule, 20
 - purely-syntactic, 1260
- q-Horn class, 29
- Q-Pos-Unit-Resolution (Q-PU-Res), 1149
- Q-resolution, 1146, 1179–1180, 1205, 1210, 1212, 1360
 - dependencies, 1183
 - function extraction, 1205
 - Q-Pos-unit, 1149
 - Q-unit, 1148
 - round-based function extraction, 1206
 - symmetries, 1184
 - tree-like, 1211
- Q-resolution calculus, 1179
- Q-unit-resolution, 1148
- QBF, 51, 398, 402, 410, 419, 481, *see also* Boolean formula, 748, 755, 756, 781, 1135, 1177–1221
 - blocked clause, 1199
 - conflict-driven clause learning, *see* QCDCL
 - cost, 1213
 - cube learning, 1184

- decision making, 1184
- expansion-based solving, 1177
- function extraction, 1204–1207
- learning, 1184
- preprocessing, 1177, 1198–1204
- proof complexity, 1207–1215
- proof systems, 1179
- proof theory, 1177
- pure literal, 1199
- search-based solving, 1177
- semantics, 1178
- strategy extraction, 1177
- unit propagation, 1184
- variable elimination, 1199
- winning strategy, 1178, 1204–1207
- QCDCL, 1179–1191
- QG
 - FQG, 828
 - IQG, 828
 - $QGi(v)$, 827
 - QG0–QG2, 826
 - QG10, 842
 - QG10–QG12, 841
 - QG13–QG15, 842
 - QG3–QG9, 829
- QRAT, 1177, 1198–1204
 - definition, 1201
 - function extraction, 1205
- QRAT+, 1204
- QU-resolution, 1181
- qualitative class, 618
- quality of the estimate, 1003
- quantification logic, 12
- quantified Boolean formula, 51, 398, 402, 410, 419, 781, 809, 995, 1157, 1177, 1178, *see also* QBF, 1334
 - minimal falsity, 583
 - Q-HORN, 54
 - Q-resolution, 54
- quantified constraint expression, 1138
- quantified extended Horn formula, 1148
- quantified resolution asymmetric tautologies, *see* QRAT
- quantifier, 1131, 1312, 1331
 - block, 1178
 - partially ordered, 1145
- quantile weights, 1037
- quasigroup, 821
 - disjoint, 831
 - equivalent, 849
 - identity, 829
 - isotopic, 849
- quasilinear, 233
- quasipolynomial, 305
- Quine, 19
- R -operator, 276
- radio spectrum repacking, 486
- Ramon Lull, 9
- Ramsey number, 832
- random, 893
 - graph, *see* random graph
 - number, 476
 - propositions, 1346
 - reordering, 476
 - restarts, 355
 - restriction, 259
- random forest, 485, 486, 491, 493
- random graph, 258
 - edge density, 258
- random k -CNF formula, 257, 262, 277, 279, 305, 312
- random matrix, 1035
- random problems
 - $(2 + p)$ -SAT, 441
 - 2-SAT, 441
 - 3-SAT, 999
 - clustering, 444
 - frozen variables, 447
 - generative models, 447
 - k -SAT, 184
 - Max k -SAT, 440
 - random 2-SAT, 40, 44, 222
 - random 3-SAT, 222, 1345
 - random k -SAT, 33, 36, 222, 437, 872, 1336
 - solution-space geometry, 444
- random restarts, 126
- random walk, 215
 - algorithm, 451, 675, 676, 878
 - focused, 215
 - multistart, 675
 - random problems, 451

- randomization, 464, 472, 475
 - branching, 146
 - deterministic, 477
- randomized
 - algorithm, 679
 - block, 1331
 - literal, 1332
 - variable, 1331
- randomized flooding, 534
- randomized rapid restarts (RRR), 472
- randomized reordering, *see* renaming
- randomized tie-breaking, 476
- range allocation, 1280
- rank-width, 721
- ranking function, 814
- rapid randomized restart, 473
- rapid restarts, 472
- RAT, 410–413, *see also* resolution asymmetric tautology, 1200
- RAT inference, 313, 314
- rational magic square, 844
- re-encoding proof, 655
- reactive scaling and probabilistic smoothing (RSAPS), 219
- read-once refutation, 576
- reason, 137, 1243
- reason constraint, 290, 291
 - reduced, 292
- reason partition, 1109
- reasoning
 - algebraic, 416, 419
 - autarky, 190, 204
 - circuit-level, 416
 - parity, 414, 418, 419
 - symmetry, 322
- reconstruction stack, 412, 413
- rectangular approximation, 1037
- recursive function, 14
- recursive learning, 417
- recursive minimization, 145
- reduce, 394, 407
 - Latin square, 849
- Reduced Boolean Circuits (RBC), 1051
- reduced cost, 965
- reduced deficiency, 620
 - maximal, 623
- reduced ordered binary decision diagram (ROBDD), 1362
- reduction, 355, 358, 372, 683
 - autarky, 588, 589, 623
 - cone-of-influence, 393, 409
 - counting, 995
 - equality, 127, 1001
 - equivalence, 393, 404
 - fixed-parameter tractable, 726
 - forall, 1136, 1361
 - future, 374
 - generalised unit-clause propagation, 371
 - look-ahead, 372
 - look-ahead reduction, 373
 - monotone-input, 409
 - parsimonious, 995
 - randomized, 685
 - SERF, 683
 - to SAT, 233, 601
 - universal, 1136, 1179, 1182
 - volume, 488
- reduction algorithm, 291, 294, 295
- redundancy
 - propagation, 642, 643
 - set-propagation, 642
- redundant, 635, 637
 - generator, 519
- redundant clause, 398, 403, 407
- redundant fault, 1067
- refinement, *see also* partition, refinement, 1272
 - abstraction, 811
 - Boolean, 1232
 - ordered partition, 527
 - partition, 552
- refutation, 751, 753, 1192
 - completeness, 102, 1052
 - read-once, 576
 - regular, 22
 - resolution, 1179
 - wrong, 1180
- refutation length, 260
- refutation size, 260
- refutation system, 238
- refutation width, 260
- regression, 485, 490

- regular proof, 646
- regular resolution, *see also* resolution,
 - regular, 259
- RegWRTI, 254, 255
- reinforcement learning, 498
- relation
 - accessibility, 1228
 - affects, 775
- relative marginal contribution (RMC), 495
- relaxable clause, 905
- relaxation variable, 940
- renaming, 476, 1152
 - CNF formula, 698
- replica method, 868
- representation, 576
 - graph, 1357
 - in SAT, 773
 - product-of-maxterm, 513
 - sum-of-minterm, 513
- residual formula, 442
- residual function, 561
- $Res(k)$, *see* k -DNF resolution
- resolution, 19–21, 102, 234, 235, 238, 395, 399, 401, 405, 638, 671, 1164, 1179, 1192, 1198, 1207, 1208
 - as depth-0 LK refutation, 318
 - asymmetric tautology, *see* resolution asymmetric tautology (RAT)
 - autarkies, 589
 - binary, 126, 127, 405
 - bounded, 671
 - chain, 638, 646
 - clause resolution, 1164
 - completeness, 240, 913
 - complexity, 21, 23, 443
 - directional, 105, 1349
 - extended, 20, 234, 235, 251, 312, 409, 639, 651
 - extended resolution, 626
 - extension rule, 315
 - fusion rule, 297
 - generalized, 285, 287, 308
 - ground, 20
 - hyper, 127
 - binary, 393, 396, 405–407, 417, 644
 - input, 241
 - k -DNF, 319
 - length, 253, 255, 267
 - length-width lower bound, 259, 260
 - long-distance, 1180
 - low, 968
 - MaxSAT, 913, 942
 - polynomial calculus, 275
 - pool, 254
 - proof DAG, 239
 - proof system, 238
 - quantified Boolean, 1146
 - random problems, 443
 - refutation, 21, 239
 - regular, 22, 240, 259
 - separation, 240
 - regular input, 646
 - RegWRTI, 254
 - resolution rule, 239
 - resolvent, 239
 - rule, 239, 395, 638
 - self-subsuming, 399
 - separation, 240, 263, 264
 - from cutting planes, 302
 - from polynomial calculus, 275
 - from tree-like resolution, 261
 - singular DP-resolution, 576
 - size, 255
 - size-width lower bound, 259, 260
 - soundness, 240
 - space, 253, 255, 267
 - symmetry, 322
 - term, 1166
 - term resolution, 1166
 - total space, 262
 - trace, 103, 654
 - trade-off, 264
 - length-space, 266, 268
 - length-width, 267
 - size-space, 266
 - size-width, 267
 - space-width, 265
 - tree, 21
 - tree-like, 239

- trivial, 240
- unit, 103, 115, 241, 395
- usable clauses, 589
- weakening rule, 240
- width, 255, 259, 262, 267
- with subsumption, 671, 682
- resolution asymmetric tautology, 410–413
- resolution asymmetric tautology (RAT), 313, 410, 641
- resolution barrier, 418
- resolution proof, 392, 393, 397, 415, 418, 750
- resolution proof system, *see* resolution
- resolution refutation, 1179
- resolution-path dependency scheme, 1183, 1202
- resolution-subsumed clause, 410
- resolvent, 19, 102, 202, 239, 395, 590, 638, 646, 671, 750, 1165
 - autarky, 205
 - look-ahead, 205
 - constraint, 203, 204
 - double look-ahead, 207
 - hyper binary, 406
 - look-ahead autarky, 205
 - outer, 1201
 - subsuming, 399
- resolving, 102
- restart, 24, 126, 247, 251, 254, 255, 355, 465, 472
 - adaptive, 247, 269
 - policy, 241
 - random, 355
 - rapid randomized, 473
 - SMT, 1289, 1294
 - warm, 155
- restricted system, 606
- restricted versions, 593
- restriction, 699, 931
 - random, 259
- result certificate, 127
- reverse unit propagation, 128, *see also* RUP, 244, 251, 313, 408, 638, 649
- rewriting, 417
- ridge regression, 490
- Riss3g, 487
- ROBDD, 743
- robust algorithm, 701
- root level, 411
 - assigned, 411
 - forced, 411
 - inconsistent, 411
- rooted tree, 352
- roots, 396, 406
- RRR, *see* rapid restarts
- RS, 410
- RSat, 401
- RSat solver, 401
- rule
 - asymmetric tableau, 407
 - axiom, 1191
 - branch/merge, 111
 - clause elimination, 1199
 - clause modification, 1200
 - clauses addition, 1199
 - cut, 317, 914
 - dilemma, 110
 - extension, 312, 315, 639
 - freebie, 217
 - inference, 109
 - instantiation, 1207
 - Jeroslow-Wang, 377
 - two-sided, 377
 - majority, 24
 - multiplication, 1096
 - negation, 1096
 - resolution, 239, 395, 638
 - shortest-clause, 24
 - star, 910
 - strengthening, 296, 297
 - subsumption, 19
 - symmetry, 1184, 1214
 - transformation, 670, 681, 909
 - transition, 1293–1295, 1301
 - unit-clause, 136, 394, 405, 637, 1103
- running time, 486
- runtime variation, 463, 465
- RUP, *see also* reverse unit propagation, 251, 408, 1201
 - clause, 314

- proof, 251, 313
- saddle point, 220
- safe assignments, 624
- safe partial assignment, 624, 625
- safety, 741, 747
 - checks, 1009
 - property, 804
- SAPS, 492
- SAT, 670
 - 2-SAT, 672
 - algorithm, 672, 676
 - 3-SAT, 679
 - algorithm, 679
 - upper bound, 679, 686
 - k -SAT, 670, 674, 685, 686
 - algorithm, 674–679, 684
 - upper bound, 674–676, 679, 685, 686
 - k -SAT- f , 670
 - upper bound, 685
 - algorithm, 32, 127
 - symbolic, 107
 - CDCL, 152
 - competition, 269
 - general, 670
 - general SAT, 670, 679
 - algorithm, 679–681
 - upper bound, 679, 681, 685, 686
 - incremental, 153, 392, 403, 412, 413, 943
 - parameterized, 700
 - problem, 133, 233
 - representation, 773
 - SAT- f , 670, 686
 - upper bound, 685
 - solver, *see* SAT solver
 - symbolic, 107
 - translation, 1118
 - unique k -SAT, 670, 679, 685
 - algorithm, 679
 - upper bound, 679, 685
- SAT competition, 184, 487, 490, 491, 493, 494, 496, 498, 636, 660
- SAT modulo theories (SMT), 779
- SAT oracle, 1030
- SAT solver, 136, 152, 158, 234, 657
 - OKsolver, 603
 - complete, 603
 - march, 603
 - modoc, 599
 - proof logging, 249
 - UnitMarch, 603
- SAT sweeping, 417, 418, 754
- SAT-based ATPG, 1071
- sat13, 484
- SatELite, 392, 393, 398, 399, 402, 412
- SATenstein, 487, 500
- satisfaction, 16
 - constraint, 481, 819
- satisfaction-driven clause learning (SDCL), 315, 645
- satisfiability, 4, 1334, 1336
 - balanced linear, 614
 - Boolean, 1064, 1069
 - Boolean circuit, 1050
 - estimator, 372, 380
 - existential quantification, 104
 - index, 29
 - inference rule, 109
 - Max-2-SAT, 44
 - Max-SAT, 43, 45
 - modal, 1223
 - model, 1142
 - modulo a theory (SMT), 1268, 1271
 - modulo SAT, 851
 - modulo theories, 851
 - non clausal, 1172
 - parameter, 693, 694, 701, 721
 - hybrid, 721
 - parameterized, 701
 - problem, 615
 - proof, 1202
 - pseudo Boolean, 1363
 - quantified 2-CNF, 1153
 - quantified Boolean, 1132, 1137, 1157, 1159, 1334, 1360
 - quantified Horn, 1151
 - satisfiability index, 29
 - stochastic, 783, 1331

- satisfiability algorithms
 - incomplete, 213
- satisfiability equivalent formula, 1133
- satisfiability threshold, 37, 40, *see also*
 - threshold, 438, 439
 - order of, 41
- satisfiability-preserving clause, 624
- satisfiability-preserving partial assignments, 624
- satisfiable, 4, 237, 637, 670, 746, 1050
 - CNF formula, 699
 - formula, 637, 775, 889, 1133
 - propositionally, 1233
 - regime, 457
- satisfy, 236, 237
- satisfying assignment, 136, 411, 798, 1015
- SATO, 133
 - HQG1, HQG2, 829
 - IQG0–IQG2, 829
 - QG1, QG2, 827
 - QG13, 843
- saturated count, 1021, 1028
- saturating, 914
- saturation, 110, 286, 287, 308
 - rule, 1097
- Satz, 416
- SATzilla, 490, 492–494, 497
- saucy, 533–535, 548, 565
- SBP, *see* symmetry breaking predicate
 - partial, 544
- SBS, 494, 495
- scalable, 269
- scaling and probabilistic smoothing (SAPS), 219
- scenario tree, 1355
- Schöning’s algorithm, 676, 879
 - derandomization, 677
- Schaefer’s dichotomy theorem, 672
- scheduling, 92, 936
- scheme
 - 1UIP, 124
 - branching, 372
 - practical algorithms, 374
 - theoretical algorithms, 373
 - dependency, 1213
 - two-watched-literals, 298
- scope, 1158
- search, 1161
 - backtrack, 465, 475
 - branch and bound, 906
 - depth-first, 1338, 1348
 - galloping, 1028
 - greedy, 215
 - heuristics, 883
 - local, 42, 90, 213, 484, 485, 491, 492, 497, 603, 1353, 1358, 1363
 - stochastic, 213
 - proof, 644
 - semantic branching, 1240
 - syntactic branching, 1240
 - systematic, 1338, 1346, 1348
 - tree, 528
 - bounded, 696
 - permutation, 549, 550
- search problem, 300
- search space splitting, 970
- search-based solving, 1179–1190
- second moment method, 447, 449
- security, 937
- selection
 - algorithm, 481, 489, 494, 496, 498, 500
 - per-instance, 489
 - problem, 489
 - variable, 153
- selector
 - function, 300
 - index, 539
 - per-instance algorithm, 490
 - variable, 300, 301
- self-orthogonal, 826
- self-reducible, 704, 1028, 1029
- self-subsuming resolution, 399
- self-subsumption, 255
- SEM, 849
- semantic
 - bit-vector, 791
 - bounded, 742
 - branching, 1240
 - symmetry, 525, 561
 - tree, 21

- semi-direct product, 523
- semi-maximal autarky, 594
- semialgebraic proof system, 271, 311
- semicentral class, 616
- semidefinite programming, 45
- separation, 240
- separator, 1002
- sequence, 1293
- sequent, 316
 - calculus, 316, 1215
 - initial, 317
- sequential plans, 767
- sequential testing, 777
- SERF, 683
- SERF-completeness, 683
- set
 - backdoor, 704
 - clause, 586
 - finite, 598
 - independent, 577
 - commutatively generating, 596
 - commutatively semi-generating, 596
 - complete, 1237
 - conflict, 118, 120
 - extremal, 399
 - hitting, 698
 - hole, 828
 - independent, 697
 - maximum, 1114
 - large, 831
 - minimal unsatisfiable, 635
 - permutation, 549
- set enumeration, 159
- set-blocked clause, 410, 641
- set-propagation redundancy, 642
- sgen formula, *see* subset cardinality formula
- Shannon's expansion, 18, 31
- sharp P, *see* #P
- sharp threshold, 40
- Sherali-Adams proof system, 311
- short conjugate-orthogonal identity, 829
- Short Resolution Refutation (SRR), 576
- shortest-clause rule, 24
- Shostak's method, 1298
- sibling pruning, 1000
- sign-central matrix, 619
 - minimal, 619
- signal correlation, 1063
- signature, 398, 1269
- signed permutation, 560
- simple linear autarky, 609
- simple path constraint, 748
- simple pure autarky, 607
- simple rules, 110
- simulate polynomially, 237
- simultaneous swap, 510
- single, 995
 - algorithm, 687
 - clause, 614
 - permutation, 555
- single best solver (SBS), 495
- Single Step Tableau (SST), 1257
- singular DP-resolution, 575
- size
 - cutting planes, 302
 - encoding, 91, 944
 - estimating tree, 362
 - formula, 236, 237
 - input, 694
 - Nullstellensatz, 271
 - polynomial calculus, 274
 - proof, 237
 - resolution, 239, 255
 - v.s. degree, 276
 - v.s. width, 260, 261
- size-cost-capacity theorem, 1215
- size-degree trade-off, 281
- size-space trade-off, 266, 281, 306, 307, 319
- size-width
 - lower bound, 259, 260
 - technique, 1210
 - trade-off, 267
- Skolem, 15
- Skolem function, 1177, 1178, 1191, 1205
- slack, 288
 - subadditivity, 292, 294
- Slaney, 831
- SLS, *see* stochastic local search

- SLUR, 28, 395
- SMAC, 485, 487, 488, 494
- Small Unsatisfiability Subset (SUS), 576
- small-domain encoding, 1280, 1310
- smoothed form, 1002
- SMT competitions, 484
- SMT solver, 500, 803, 1268, 1285
 - lazy, 1285
- SNNAP, 493, 494
- SNS-matrix, 618
- social golfer problem, 841
- soft cardinality constraints, 157
- soft clause, 905
- software development, 483
- software verification, 391, 500, 1159
- solution, 411, 1015
 - numerically least, 537
 - reconstruction, 411
 - state, 553
 - symmetry, 564
 - tree, 1332
- solution backjumping, 1167
- solution counting, *see* model counting
 - approximate, 160
 - exact, 160
- solution density, 91, 93
- solution distribution plot, 198
- solution reconstruction, 411
- solution sampling, 1003, 1005
- solver
 - 2QBF, 1018
 - Böhm, 189
 - backup, 490, 491
 - CDCL, 392, 393, 397, 415, 500, 644
 - CDCL SAT, 133, 140, 151, 152, 155, 160, 161, 163, 418
 - nondeterministic, 252
 - SAT, 136, 152, 158, 657
 - complete, 603
 - theory, 1285
- solver subset selection, 491
- solving
 - assumption-based SAT, 942
 - SAT, 605
 - practical, 653
- SOP, *see* sum-of-products
- sorting literals, 652, 653
- sound, 909, 1052
 - abstraction, 1311
- soundness, 237, 240, 1295, 1306
- SP, *see* survey propagation
- space, *see* clause space, *see also* line space, *see also* total space
 - cumulative, 266, 268
 - cutting planes, 302, 304
 - of vertex permutations, 548
 - polynomial calculus, 274, 279
 - resolution v.s. polynomial calculus resolution, 275
 - v.s. degree, 280, 281
 - v.s. width, 263, 264
- space-degree trade-off, 280
- space-width trade-off, 265
- Sparkle, 495
- Sparkle SAT Challenge, 494, 500
- sparse encoding, 78
- sparse multinomial regression, 491
- sparse structure, 1282
- sparsification, 684
- sparsity, 530
- Spear, 484
- spectral analysis, 39
- sphere covering bound, 677
- Splatz, 400
- splitting, 573, 574
 - disjunctive, 574
 - unique, 574
- splitting tails, 1037
- spurious
 - counterexample, 811
 - dependency pairs, 1182
 - traces, 811
 - transition, 809, 811
- square hypergraph, 614
- square-free, 270
- SSAT, 783
- stabbing planes, 306
- stabilizer, 525
- stable
 - coloring, 527
 - distribution, 466

- formula, 583
- marriage, 88
- stack
 - decision, 137
 - reconstruction, 412, 413
- stamping, 396, 399
- standard dependency scheme, 1182
- standard deviation, 1023
- standardised matrix, 611
- Stanley Jevons, 11
- star rule, 910
- state, 1346
 - conflict, 553
 - decision, 553
 - error, 795
 - goal, 1346
 - initial, 1346
 - solution, 553
- static learning, 1290
- static single assignment, 801
- static symmetry breaking, 557
- statistical framework, 1009
- statistical physics, 223, 224, 859
- Steiner pentagon system, 841
- Steiner system, 840
 - quadruple, 840
 - triple, 840
- Stepped Reckoner, 10
- Steven Cook, 21
- Stickel, 831, 842, 849, 850
 - QG2(12), 827
- stochastic
 - local search, 42, *see also* local search, 1064, 1353
 - satisfiability, 783, 1331
 - variables, 887, 1354
- stochastic constraint
 - programming, 1354, 1362
 - satisfaction program (SCSP), 1354
- stochastic optimization programming
 - language, 1355
- Stoics, 8
- strategy, 1143, 1178
 - branching, 384
 - conflict-driven, 183
 - decision, 247, 254
 - hybrid, 1008
 - look-ahead, 183
 - universal, 474
 - winning, 1178
- strategy extraction, 1211
- streamlining constraints, 844, 1007
- strengthening, 399, 402, 407–409, 412, 1200
 - invariant, 755
- strengthening rule, 296, 297
- strict
 - domination, 702
 - problems, 701
- strong backdoor, 471
 - set, 704
- Strong Exponential Time Hypothesis (SETH), 233
- strong hypergraph-colouring, 626
- strongest post-condition, 813
- strongly 2-universal sub-family, 1025
- strongly connected component, 396, 404, 406
- strongly symmetric matrix, 845
- strongly-balanced autarky, 626
- structural hashing, 416, 417
- structure, 4, 143
 - algebraic, 4
 - community, 723, 1357
 - cycle, 540, 560
 - data, 913, 1356
 - Kripke, 741, 1226
 - minimal monotonic, 41
 - Montague-Scott, 1229
 - sparse, 1282
- structure-based preprocessing, 414
- Structured Procrastination, 488
 - with Confidence, 489
- stuck-at fault, 1066
- Stuck-At Fault Model (SAFM), 1066
- stuck-at-0 (SA0), 1066
- stuck-at-1 (SA1), 1066
- Stålmарck algorithm, 109, 1048
- Stålmарck's procedure, 417
- Stålmарck's proof procedure, 203, 1048
- sub-block, 1331
- sub-clause-set
 - autark, 589
 - largest, 589

- sub-family
 - strongly 2-universal, 1025
 - uniform, 1025
- sub-solver, 470
- subadditive, 292, 294
- subexponential
 - function, 683
 - reduction family, 683
- subformulas
 - disjunctive, 1240
- subgroup, *see* group, subgroup
- submodular function, 497
- subproblems, 351, 354
- subset, 560, 775
 - autark, 589
 - autarky, 594
 - cardinality-minimal, 158
 - small unsatisfiable, 726
- subset cardinality formula, 259, 277, 309, 310
- subset minimal subsets, 159
- subset propagation redundancy (SPR), 315
- substituted pebbling contradictions, 281
- subsume clause, 236, 240
- subsuming resolvent, 399
- subsumption, 19, 393, 395, 397, 402, 403, 412, 670, 682, 1199, 1230
 - backward, 398–400, 402
 - clausal, 102
 - clause, 393
 - forward, 397, 399, 400
 - global, 399
 - on-the-fly, 399
 - rules, 19
- subtree
 - minimal false, 1164
 - minimal true, 1166
 - mutually consistent, 1240
 - mutually-inconsistent, 1241
- success-to-failure transition, 883
- sum constraints, 1281
- sum term, *see also* implicant, 513
- sum-of-minterm representation, 513
- sum-of-products, *see also* disjunctive normal form, 514
- sum-of-squares proof system, 311
- summary edge, 809
- supervised learning, 490, 493
- support, 1015, 1020
 - clause, 79
 - independent, 1026
- survey propagation, 224, 454, 891, 1359
- surveys, 1360
- survival analysis, 491
- sweeping, 417
- switching lemma, 320
- syllogism, 8
- symbolic SAT, 107
 - solving, 104, 107, 108
- symbolic simulation, 810
- symmetric function, 743
- symmetric group, 520
- symmetric magic square, 844
- symmetry, 85, 88, 92, 93, 127, 152
 - breaking, 537, 544, 642, 644, 650
 - dynamic, 559
 - full, 543
 - partial, 543
 - predicate, 152, 537
 - static, 557
- classical, 562
- CNF, 525
- composition table, 511
- conditional, 558, *see also* symmetry, local
- constraint, 564
- detection, 531
- first-order, 562
- global, 558, 559
- hierarchical, 562
- higher-order, 562
- in Boolean satisfiability problems, 562
- local, 558, 559
- mixed, 526
- rule, 1184, 1214
- semantic, 525, 561
 - partial, 561
 - total, 561
- solution, 564
- syntactic, 525

- tree, 563
- value, 526, 564
- variable, 526, 564
- symmetry composition table, 511
- symmetry elimination, 848
- symmetry reasoning, 322
 - DRAT, 323
 - dynamic, 323
 - proof complexity, 323
 - static, 323
- symmetry-breaking predicate (SBP), 511
- syntactic
 - branching, 1240
 - symmetry, 525
- synthesis, 417, 937
- system
 - autarky, 605, 606
 - natural, 608
 - axiom, 3
 - Frege, 235, 316, 322, 1214
 - Kirkman triple, 840
 - normal autarky, 606
 - proof, 237, 238, 639, 1178
 - restricted, 606
 - Steiner, 840
 - pentagon, 841
 - quadruple, 840
 - triple, 840
 - transition, 792
- table
 - function, 403
 - symmetry composition, 511
- tableau, 745, 751, 1052
 - branch
 - complete, 1052
 - contradictory, 1052
 - open, 1052
 - closed, 1052
 - cut rule, 1048, 1052
 - finished, 1052
 - non-branching rules, 1052
 - refutation, 1052
- tableau calculus, 599, 1047
- tableaux procedure, 1235
 - modal logics, 1235
- tail function, 1036
- target
 - cell, 528, 551
 - vertex, 551
- tau-function, 359
 - bound, 360
- tau-method, 365
- tautological, 236
 - clauses, 407
- tautology, 5, 402, 407, 409, 1179
 - asymmetric, 407, 408, 410, 1201
 - quantified, 1201
 - elimination, 402, 1199
 - hidden, 408
 - proof, 655
 - resolution asymmetric, 410
- technique
 - advanced probing, 393, 407, 408
 - approximation, 1359
 - lower-bound, 1210
 - meta-algorithmic, 481, 496, 499
 - size-width, 1210
 - threshold pruning, 1338
 - unit resolution, 115
- technology mapping, 417
- temporal logic, 740
- temporal planning, 779
- term, 270, 1166, 1178
 - ground, 1269
 - resolution, 1166
 - sum, 513
- termination, 814
- termination tree, 114
- test case generation, 739
- test pattern generation, 1064
 - automatic, 1064
- testable fault, 1067
- theorem
 - Cayley's, 521
 - Dichotomy, 1139
 - Gödel's incompleteness, 14
 - Herbrand's, 15
 - parenthesis, 408
 - Schaefer's dichotomy, 671, 672
 - size-cost-capacity, 1214
- theory, 1271
 - arithmetic, 1273, 1310

- arrays, 795, 796, 1274, 1276, 1300, 1301, 1303
- bit-vectors, 795, 797, 1274, 1285, 1306, 1312
- combining, 1271, 1299, 1303
- completeness, 694
- computational complexity, 322
- conflict, 1286, 1288, 1291, 1293, 1296, 1308, 1309
- convex, 1271, 1298, 1307
- deduction, 1288
- empty, 1273
- equality, 1273, 1280
- equality with uninterpreted functions, 1273
- expansion, 1271
- inductive data types, 1275
- layered solvers, 1302
- lemma, 1271, 1296, 1297
- matching, 1313
- model, 4, 15
- pointers, 795
- rewriting-based solvers, 1302
- Shostak, 1298
- solvers, 1285, 1286, 1297
- stably infinite, 1306
- theory propagation, 1288, 1291, 1294, 1296
- threshold, 39, 222, 1023, 1028
 - coarse, 40
 - completeness, 802
 - probability, 1354
 - satisfiability, 40, 438, 439
 - sharp, 40
- threshold pruning technique, 1338
- tiling problem formula, 258
- time-frame, 799
- tolerance, 1018, 1020, 1023
 - additive, 1018
 - multiplicative, 1018
- top-level atom, 1231
- torso graph, 722
- total
 - assignment, 236, 637, 699, 1292, 1295
 - truth, 699, 1231
 - space, 262, 304
 - total satisfying assignment, 586
 - totally symmetric function, 560
 - towers of Hanoi, 90
 - trace, 646, 798
 - spurious, 811, 812
 - tracecheck, 645
 - tractability, 1334
 - fixed-parameter, 694, 695
 - trade-off, 264, 280
 - k -DNF resolution, 319
 - cutting planes, 306, 307
 - length-space, 266, 268
 - length-width, 267
 - Nullstellensatz, 281
 - polynomial calculus, 280, 281
 - resolution, 265–268
 - size-degree, 281
 - size-space, 266, 281, 306, 307, 319
 - size-width, 267
 - space-degree, 280
 - space-width, 265
 - transformation, 939
 - Boolean algebra, 76
 - CNF, 75
 - CNF to 3-CNF, 78
 - Tseitin encoding, 76
 - transformation rule, 670, 681, 909
 - black and white literals, 682
 - blocked clauses, 682
 - bounded resolution, 671
 - elimination by resolution, 671, 682
 - resolution with subsumption, 671, 682
 - subsumption, 670, 682
 - unit clause elimination, 670, 681
 - transition
 - condensation, 446
 - dynamical, 446
 - phase, 37, 93, 152, 222, 438, 446, 454, 861, 869, 870, 872, 999
 - spurious, 809, 811
 - success-to-failure, 883
 - transition relation, 798
 - transition rules, 1293–1295, 1301
 - transition system, 792

- transitive reduction, 407
- transitivity constraints, 1283, 1284, 1290
- translation
 - base, 773
 - constraint, 1118
 - Cook, 322
 - Paris-Wilkie, 322
- translational approach, 1224
- traps, 221
- traveling salesman problem (TSP), 859
- tree, 362
 - adorning, 363
 - AND-OR, 1339
 - assignment, 1332
 - backdoor, 705
 - bounded search, 696
 - canonical tree probability distribution, 364
 - decision, 493, 1337, 1338
 - binary, 1337
 - canonical, 1337
 - distance, 365
 - enumeration, 355
 - implication, 208
 - measure, 365
 - MINIMAX, 1339
 - policy, 1355
 - probability distribution, 363
 - resolution, 21
 - rooted, 352
 - scenario, 1355
 - search, 355, 528
 - permutation, 549, 550
 - semantic, 21, 1163
 - size estimations, 364
 - solution, 1332
 - symmetry, 563
 - tau-method, 365
 - termination, 114
 - variance, 364
- tree decomposition, 713, 714
 - algorithms, 719
 - width, 713
- tree search model, 468
- tree-based look-ahead, 208, 209
- tree-like resolution, 239
- treeSAT formula, 1353
- treewidth, 712, 713, 720
 - conflict, 718
 - consensus, 718
 - dual, 713
 - expression, 718, 719
 - incidence, 713
 - modular, 720
 - primal, 713
- trie, 406, 409
- trigger, 1313
- triggering, 1248
- trivial autarky, 588
- trivial dependency scheme, 1182
- trivial proof, 646
- trivial resolution, 240
- trivial subgroup, 518
- true, 1158, 1162, 1163, 1170
- trust management (TM), 1336
- truth, 16
 - assignment, 236, 513, 698, 699, 931, 1049, 1231
 - consistent, 1049
 - extension, 1049
 - partial, 1231
 - restricted, 1232
 - restriction, 931
 - satisfying, 1050
 - total, 699, 1049, 1231
 - logical, 5
 - value, 512
- Tseitin encoding, 76, 91, 93, 416, 417, 797
- Tseitin formula, 256, 262, 275, 276, 279, 281, 305, 315, 320–322, 415
- Tseitin procedure, 581
- Tseitin transformation, 743, 746
- tuple, 842
 - branching, 357, 362, 368
- Turing machine
 - halting problem, 233
 - probabilistic, 1017
- twin variables, 274, 275
- two-player game, 1211
- two-sided approximation, 1019

- two-watched-literals scheme, 298
- UIP literal, 244
- undefined value, 135
- unhiding, 408
- uniform
 - clause-set, 617
 - distribution, 1005
 - sub-family, 1025
- uninterpreted functions, 796
- union, 516
- union find, 412
- unique implication point, 124, 143, 1108, 1167
- unique minimal unsatisfiable formula, 578
- unique polynomial representation, 50
- unique sensitization points (USP), 162
- unique splitting, 574
- unique variable, 608
- unit, 1166, 1186, 1339
 - constraint, 1103
 - literal, 1186
 - partition, 515
 - resolution, 103, 115, 241, 395
- unit clause, 20, 103, 185, 241, 394, 395, 403, 405, 407, 408, 637, 670, 883, 1270, 1339
 - elimination, 670, 681
 - rule, 136, 1103
- unit propagation, 36, 92, 115, 136, 140, 185, 241, 242, *see also* reduction, 393–397, 405–408, 411, 415, 416, 451, 637, 658, 699, 702, 707, 883, 1103, 1136, 1161, 1338, 1339
 - BCP, 208
 - Boolean constraint propagation (BCP), 136
 - quantified Boolean, 1136
 - random problems, 456
 - reverse, 638
 - two-watched-literals scheme, 298
- unit-clause propagation
 - autarkies, 603
- unit-clause rule, 20, 394, 405, 637
- universal
 - block, 1333
 - player, 1178
 - variables, 1333
 - expansion, 1361
- universal expansion, 1199, 1203
- universal reduction, 1136, 1179, 1182, 1200, 1202, 1205, 1214
 - dependency, 1183
- universal strategy, 474
- unobservability, 77
- unsatisfiability
 - minimal, 571, 578–581, 583
 - problem, 615, 622
 - problem, 615, 616
 - proof, 1202
- unsatisfiable, 237, 1050
 - formula, 637, 887, 1133
 - minimal, 725
 - minimal, 571
 - propositionally, 1272
 - subset, 154
 - small, 726
- unsatisfiable core, 152, 154, 601, 659, 812, 1314
 - MaxSAT, 930
 - SMT, 1314
- unsatisfied clause, 217
- unwinding, 799
 - assertion, 802
 - loop, 800
- upper bound, 25, 998, 1009, 1113
 - MaxSAT, 907
- utility function, 1092
- UTVPI constraints, 1274, 1281
- validity, 3, 4
- valuation, 1158
- value, 137
 - incompatible, 1290
 - ordering heuristics, 384
 - symmetry, 526, 564
 - truth, 512
 - undefined, 135
- Van der Waerden certificate, 834
- Van der Waerden number, 833
- vanilla version, 263, 265
- variable, 698, 1131, 1157

- activation, 153
- active, 411
- auxiliary, 77, 92
- backbone, 1002
- binary, 512, 864
- blocking, 946
- Boolean, 49, 236, 384, 512, 669, 1088
- bound, 1132
- branching, 372, 380
- chaining, 543
- commander, 81
- contingency, 782
- cut, 1031
- decision, 1354
- degree, 617
- deletion, 705
- dependent, 90, 93
- don't care, 77
- equality, 537
- existential, 1331
- extension, 283, 287, 317
- external, 412
- forced, 186
- formal, 611
- free, 1132
- fresh, 153
- frozen, 444, 447, 873
- global, 750
- independence, 1182
 - actual, 1182
- indicator, 153
- innermost, 1169
- input, 955
- integer, 1275
- internal, 412
- level, 1161
- local, 750
- main, 300
- ordering, 116
- ordering heuristics, 109, 384
- output, 955
- pivot, 1179
- principal, 674
- propositional, 135, 512, 832, 904, 931
- randomized, 1331
- relaxation, 940
- selection, 153
- stochastic, 887, 1354
- symmetry, 526, 564
- Tseitin, 76, 86, 93
- unique, 608
- universal, 1333
 - expansion, 1361
 - weighting, 609
- variable elimination, 90, 392–394, 401–403, 408, 409, 412, 418, 1191, 1199
 - bounded, 393
- variable incidence graph (VIG), 971
- variable move to front (VMTF), 247, 254, 261
- variable selection, 1075
- variable state independent decaying sum (VSIDS), 247, 254, 261, 270
- variable width distribution, 33
- variable-byte encoding, 652
- variable-clause matrix, 611
- variable-regular clause-set, 617
- variance, 1006, 1008
- variation
 - exploiting runtime, 472
 - runtime, 463, 465
- Vazirani counter, 1034
- VBS, 492, 494, 496
- Venn circles, 12
- verification, 937
 - condition, 795
 - problem, 701
- verifying SAT result, 127
- vertex
 - dominate, 1108
 - mapped, 551
 - target, 551
- vertex cover, 696
 - formula, 258
 - problem, 695
- viable partial assignments (VPA), 1348
- virtual best solver (VBS), 492
- vivification, 393, 408, 409, 412
- VMTF, *see* variable move to front
- volume reduction, 488

- VSIDS, *see* variable state independent decaying sum (VSIDS)
- warm restarts, 155
- warning propagation, 890, 893
- watched literal, 397–399
- WBO instance (F_H, F_S) , 1124
- weak k -autarky, 602
- weak autarky, 588, 624, 625, 627
- weak backdoor, 471
 - set, 705
- weak hypergraph-colouring, 626
- weak pigeonhole principle, 256, 319, 321
- weakening, 286, 290
 - cutting planes, 286, 290
 - pseudo-Boolean, 290
 - resolution, 240
 - rule, 240
 - cutting planes, 286, 290
 - resolution, 240
- weakly-balanced autarky, 626
- weft hierarchy, 698
- weight function, 1036
- weighted binaries heuristic (WBH), 189, 193
- weighted Boolean optimization (WBO), 1123
- weighted clause, 904
- weighted CNF formula, 904
- weighted-CNF-SAT, 704
- weighting
 - clause, 218, 219, 221, 931
 - variables, 609
- whitebox testing, 499
- width, 1282
 - clause, 23, 33, 236
 - cube, 1033
 - DNF, 1033
 - logarithmic, 305
 - refutation, 259
 - resolution, 255, 259, 262, 267
 - restricted branching, 369
 - tree decomposition, 713
 - v.s. length, 260, 261
 - v.s. size, 260, 261
 - v.s. space, 263, 264
- winning move, 1193
- winning strategy, 1178
- witness, 636, 639, 641, 701, 742
 - literal, 413
 - matrix, 623
- witness-producing oracles, 158
- worst-case complexity, 893
- worst-case upper bounds
 - autarkies, 602
- wrong refutation, 1180
- XOR, 414, 415, 417, 418, 1007, 1039
- XOR constraint, 414
- XOR-CNF formula, 1139
- XOR-ification, 264
- XOR-recovery, 1032
- XOR-splitting, 1031
- XOR-substitution, 264
- XP-tractable, 698
- ZBDD, 1191
- ZDD, 403, 419
- Zermelo-Fraenkel set theory, 17
- zero-one design formula, *see* subset cardinality formula
- zero-suppressed decision diagrams, 403
- ZFC, 17
- Zhang, J., 831
- Zhu, 821
- Zvi Galil, 22

This page intentionally left blank

Cited Author Index

- Abdelwaheb, A., 1159
Abdulla, P. A., 753, 756, 1051
Abe, G., 843, 844
Abel, R. J. R., 825, 837, 841, 842
Abío, I., 80, 82, 134, 157
Abraham, J. A., 755
Abramé, A., 908, 912, 918, 919
Achá, R. J. A., 918, 936
Achlioptas, D., 36, 37, 41, 224, 277,
440–443, 445–448, 450, 453,
455–457, 864, 866, 870, 873,
884, 885, 1020, 1023, 1025,
1026, 1028, 1036
Ackermann, W., 1278, 1309
Adams, R. P., 483
Adams, W. P., 44, 311
Adler, R. J., 466
Adorf, H. M., 215
Adriaensen, S., 498
Aguirre, A. S. M., 107, 456
Aharoni, R., 30, 572, 608, 609, 725
Aho, A. V., 527, 552
Ahuja, R., 1114
Aiken, A., 803
Aïmeur, E., 938
Aisenberg, J., 318
Ajtai, M., 319
Akers, S. B., 31
Akshay, S., 1026, 1027
Al-Rawi, B., 50
Alava, M., 216, 224, 451, 881, 883
Alber, J., 43
Alekhnovich, M., 235, 240, 251, 257,
260, 262, 267, 274–276, 279,
319, 442, 451, 881
Alizadeh, B., 937
Allender, E., 673, 1335
Alon, B., 321
Alon, N., 703, 864
Aloul, F. A., 43, 50, 109, 532, 534,
546, 563, 564, 644, 849,
947
Alpern, B., 801
Alsinet, T., 84, 87, 92, 906, 908, 910,
913
Altarelli, F., 894
Alur, R., 770, 1315
Alviano, M., 957, 959
Alwen, J., 266, 268
Amjad, H., 154, 635, 657, 1313
Amla, N., 660, 750, 756
Amraoui, A., 877
Anand, S., 938
Anbulagan, 43, 190, 193, 199, 213,
218, 378, 471
Andersen, H. R., 1051
Anderson, P. W., 860
Andraus, Z. S., 585, 660, 1313
Andre, P., 377
André, P., 189
Andres, B., 954
Andrew, G., 490, 496
Andrews, W. S., 843
Angell, J. B., 1067
Angluin, D., 996
Anjos, M. F., 47, 49, 917
Annexstein F. S., 28, 395
Ansótegui, C., 80, 134, 156, 158, 267,
485, 722, 914, 919, 937, 950,
953, 957, 960, 961, 965, 968,
1159, 1172
Aragon, C. R., 215

- Arai, N. H., 322
 Arai, T., 322
 Ardelius, J., 881, 883
 Areces, C., 1224
 Argelich, J., 158, 915, 917, 918, 930,
 933, 935, 936, 967
 Arif, M. F., 160
 Armand, M., 635
 Armando, A., 76, 803, 1224, 1237,
 1239, 1268, 1285, 1286, 1290,
 1291, 1302
 Arnborg, S., 714
 Arora, S., 43, 1017
 Artho, C., 660, 747, 753
 Arvind, V., 682
 Asano, T., 46
 Ashar, P., 756, 1058, 1061
 Asín, R., 81, 134, 156, 655, 944
 Aspvall, B., 27, 28, 51, 161, 403, 404,
 672, 1153
 Asteris, M., 1020, 1023, 1025, 1026,
 1036
 Atserias, A., 134, 160, 161, 251–253,
 255, 258, 260, 262, 267, 271,
 272, 275, 281, 311, 312, 319,
 726
 Audemard, G., 134, 143, 148, 153,
 160–162, 247, 269, 313, 392,
 396, 409, 564, 636, 722, 756,
 779, 1224, 1248, 1285, 1286,
 1290–1293, 1302, 1361
 Aurell, E., 216, 224, 881, 883, 890,
 892
 Auth, E., 161
 Auton, L. D., 214, 222, 1058, 1342
 Awedh, M., 750, 755
 Ayari, A., 753, 756, 1183, 1191
 Aydin, A., 1015, 1039
 Azevedo, F., 850
 Azhar, S., 1145
 Aziz, A., 109
 Baader, F., 1223, 1235, 1236
 Baarir, S., 322
 Babić, D., 483, 484, 499
 Bacchus, F., 76, 77, 92, 127, 134,
 151, 158, 251, 393, 400, 404–
 406, 644, 702, 717, 724, 914,
 918, 930, 936, 942, 957, 958,
 961–965, 993, 994, 996, 999–
 1001, 1059, 1061, 1062, 1168,
 1190, 1198, 1205, 1206, 1242,
 1335, 1344, 1349, 1357, 1360
 Bachem, A., 596
 Bachmair, L., 1179, 1273
 Bahar, R. I., 32
 Bailey, D. D., 999
 Bailey, J., 160
 Bailleux, O., 81, 92, 134, 156, 157,
 944, 945, 953, 955, 1056, 1090,
 1119–1121
 Bain, S., 213, 218, 219
 Baker, A. B., 80, 93
 Balabanov, V., 1181, 1182, 1204, 1205,
 1210, 1211, 1213
 Balas, E., 44, 1091, 1117
 Balcázar, J. L., 1028
 Balint, A., 151, 162, 487
 Ball, M., 765
 Ball, T., 805, 808–812, 1281
 Ballard, B. W., 1339
 Baluta, T., 1015, 1039
 Balyo, T., 151, 154, 162, 392, 417,
 419
 Banbara, M., 80, 158
 Bang, L., 1015, 1039
 Bannach, M., 720
 Bansal, K., 1285
 Bansal, N., 43, 44, 910
 Baptista, L., 126, 140, 147, 148, 161,
 162, 473, 474, 476, 477
 Bar-Ilan, O., 657
 Barahona, F., 49
 Barahona, P., 80
 Barak, B., 311, 1017
 Bardenet, R., 483
 Barnett, L. A., 411, 413
 Baron, M. E., 12
 Barrett, C. W., 133, 160, 797, 851,
 1274, 1275, 1285, 1286, 1290,
 1291, 1297, 1300, 1301, 1303,
 1307, 1308, 1311–1313, 1315
 Barry, J. W., 1066

- Barth, P., 50, 1090, 1112, 1114
 Barthel, W., 879, 881, 882, 895
 Basin, D. A., 753, 756, 1159, 1183,
 1191
 Bastani, O., 938
 Batchner, K. E., 156
 Battaglia, D., 890, 892
 Battiti, R., 498
 Bauland, M., 673
 Baumer, S., 879
 Baumgartner, J., 755
 Baumgartner, P., 1030
 Bayardo Jr., R. J., 116, 118, 119,
 121, 159, 161, 234, 399, 996,
 998, 999, 1242, 1342–1344,
 1358
 Bayless, S., 487
 Beame, P. W., 23, 38, 41, 119, 121,
 122, 134, 141, 159, 160, 222,
 235, 241, 251, 257–259, 265,
 266, 268, 270, 272, 277, 300,
 301, 306, 311, 319–321, 371,
 392, 400, 443, 456, 457, 646,
 889, 993, 996, 999, 1000, 1344,
 1345, 1349, 1357
 Beck, C., 265, 266, 268, 280, 304
 Beck, J. C., 936
 Becker, B., 945, 947, 1159, 1198, 1199
 Beckert, B., 914
 Beckmann, A., 322
 Beek, van, P., 384, 765
 Beerel, P. A., 32
 Beigel, R., 688
 Beihl, G., 782
 Béjar, R., 84, 87, 92
 Belabed, A., 938
 Bell, T. C., 652
 Bellare, M., 1016, 1029
 Belle, V., 1039
 Belov, A., 134, 151, 158, 162, 636,
 966–968, 1064
 Ben-Sasson, E., 23, 38, 239, 253, 259–
 265, 268, 274–276, 278, 279,
 304, 319, 330, 442, 451, 881,
 1209, 1345
 Benedetti, M., 748, 755, 1142, 1159,
 1160, 1171, 1172, 1183, 1191,
 1360
 Bengio, Y., 483
 Benhamou, B., 396, 558, 559, 562,
 849, 1089
 Bennett, F., 820–822, 825, 826, 829,
 841–843
 Bennett, P., 262, 279
 Berezin, S., 797, 1301, 1313
 Berg, J., 134, 158, 720, 930, 937, 940,
 961, 966, 967
 Bergstra, J., 483
 Berkholz, C., 265, 268, 271, 272, 280
 Berlekamp, E., 834
 Berndt, S., 720
 Berryhill, R., 158
 Bertacco, V., 417
 Berthet, C., 31
 Bertoli, P., 779, 1224, 1248, 1285,
 1286, 1290–1293, 1302
 Berzish, M., 1315
 Besnard, P., 158, 1159
 Bessière, C., 92, 155, 468
 Beth, E. W., 1025
 Beyersdorff, O., 258, 726, 1180, 1183,
 1206–1214
 Bieber, P., 937
 Biedenkapp, A., 498
 Biere, A., 32, 51, 107, 116, 133, 134,
 143–145, 147, 148, 150, 153,
 154, 157, 159–162, 244, 247,
 249, 251, 254, 282, 310, 313,
 315, 339, 392–394, 396, 398–
 403, 406–415, 417, 418, 487,
 635, 636, 640–646, 650, 651,
 653–655, 660, 661, 739–742,
 744–750, 753, 755, 756, 765,
 799, 966, 970, 1057, 1159,
 1160, 1171, 1172, 1179, 1183,
 1190, 1191, 1198–1201, 1203,
 1205, 1214, 1242, 1285, 1315,
 1360
 Biondi, F., 1015
 Birattari, M., 488
 Birnbaum, E., 996, 997
 Biroli, G., 442, 869, 872, 873, 885
 Bischoff, G. P., 755
 Bixby, R. E., 1118

- Bjesse, P., 371, 417, 753, 755, 756,
 1051
 Björklund, A., 716
 Bjørner, N., 134, 158, 1124, 1274,
 1285, 1311, 1312
 Blake, A., 19, 238
 Blinkhorn, J., 1183, 1209–1214
 Bloem, R., 1191
 Bloom, B. H., 398
 Blum, A. L., 766, 769, 770, 775, 777
 Blum, M., 483
 Bodik, R., 1315
 Bodlaender, H. L., 712, 714, 716, 719
 Bofill, M., 918, 936
 Bogaerts, B., 322
 Böhm, M., 189, 207
 Bollobás, B., 41, 224, 441, 877
 Bonacina, I., 258, 262, 279, 304, 726,
 1180, 1211, 1213, 1214
 Bonacina, M. P., 827, 831, 1302, 1303
 Bonet, M. L., 134, 158, 160, 234, 254,
 261, 263, 267, 318, 319, 323,
 722, 913, 914, 942, 950, 953,
 960, 961, 1210
 Boots, B., 1335, 1346, 1348
 Boppana, R., 321
 Borälv, A., 417, 755, 1048
 Borchers, B., 43, 44, 907, 910
 Bordeaux, L., 143, 155
 Borgs, C., 41, 224, 441, 877
 Boros, E., 29, 50, 708, 1087, 1090,
 1093
 Borosh, I., 1279, 1282, 1283
 Bose., R. C., 824
 Boudane, A., 156
 Boudou, J., 658
 Boufkhad, Y., 40, 81, 92, 134, 156,
 157, 189, 224, 257, 377, 440,
 944, 945, 953, 955, 1056, 1090,
 1119–1121
 Bouhmala, N., 214
 Bourbaki, N., 591, 611
 Boute, R. T., 31
 Boutilier, C., 1348
 Bova, S., 719
 Boy de la Tour, T., 1047
 Boyer, R. S., 1268
 Bozkurt, H. F., 498
 Bozzano, M., 756, 1285, 1286, 1290–
 1292, 1302, 1303, 1307–1309
 Brace, K. S., 31, 755
 Bradley, A. R., 159, 661, 750, 755
 Bradley, P., 878
 Brady, B., 798, 1274, 1285, 1311
 Brafman, R. I., 397, 398, 769, 1242
 Brakensiek, J., 636, 645, 655
 Brand, D., 1066
 Brand, S., 1223, 1224
 Branigan, S., 530
 Braud-Santoni, N., 1191
 Braunstein, A., 224, 225, 383, 875,
 890, 892, 893, 1359
 Brayton, R. K., 108, 109, 133, 417,
 561, 750, 754, 770, 1064, 1069,
 1071, 1072, 1074, 1102, 1115
 Breiman, L., 486
 Bresciani, P., 1235, 1236
 Breuer, M. A., 1065
 Brglez, F., 213
 Brickenstein, M., 281, 418
 Broder, A. Z., 38, 224, 453, 867
 Brown, C. A., 33, 34, 222, 559, 562
 Brown, F. M., 19
 Brown, S. D., 1159
 Brualdi, R. A., 611, 614, 617, 618,
 620
 Brueggemann, T., 26, 679, 686
 Brummayer, R., 417, 654, 660, 1285
 Bruni, R., 585
 Bruttomesso, R., 657, 1274, 1285, 1286,
 1290–1292, 1302, 1303, 1307–
 1309
 Bruynooghe, M., 322
 Bryant, R. E., 31, 32, 107, 108, 311,
 403, 534, 740, 798, 809, 1001,
 1048, 1159, 1171, 1256, 1268,
 1274, 1275, 1278, 1279, 1282–
 1285, 1290, 1311
 Bubeck, U., 54, 1144, 1145, 1152,
 1171, 1198, 1199
 Büchi, J. R., 809
 Buhrman, H., 687
 Bullen, P. S., 360
 Bultan, T., 1015, 1039

- Bunte, K., 937
 Burch, H., 530
 Burch, J. R., 32, 108, 745
 Buresh-Oppenheim, J., 45, 272, 275, 726
 Buro, M., 24, 189
 Busquets, D., 918
 Buss, S. R., 134, 160, 235, 251, 254, 272, 276, 303, 312, 315, 318–323, 641, 643, 914, 1209
 Büttner, M., 772
 Bylander, T., 765

 Cabiscol, A., 84, 87, 92, 918, 935
 Cabodi, G., 750, 755
 Cadar, C., 797, 803
 Cadoli, M., 157, 755, 1159, 1162, 1168, 1184–1186, 1199, 1200, 1338, 1342
 Cai, L., 696
 Cai, S., 401, 968
 Calabro, C., 233, 680, 681, 684–686
 Cammann, S. V. R., 843
 Campigotto, P., 498
 Canfield, E. R., 516
 Cao, N., 843
 Cao, S., 938
 Caridroit, T., 134, 158
 Carlier, C., 377
 Carlier, J., 189
 Carlson, D. A., 265
 Carter, J. L., 836, 1016, 1025
 Casas-Roma, J., 916
 Castellini, C., 1159, 1224, 1239, 1268, 1285, 1286, 1290, 1291
 Castelluccia, C., 150, 162, 414, 1016, 1030
 Ceder, G., 938
 Celoni, J. R., 264, 265
 Ceria, S., 44, 1117
 Cerna, D., 411, 413
 Cesati, M., 698
 Cha, B., 84, 92, 213, 218
 Chai, D., 50, 287, 290, 291, 296, 298, 1111, 1112, 1114
 Chai, K. M. A., 418
 Chaki, S., 808
 Chakraborty, S., 159, 1016–1020, 1023, 1025, 1027, 1028, 1030, 1032, 1034, 1036–1039
 Chambers, B., 417
 Chandrasekar, K., 1360
 Chandrasekaran, R., 28
 Chandru, V., 28, 1284, 1362
 Chang, Y., 843
 Chao M.-T., 24, 36, 37, 437, 438, 452, 883
 Chatalic, P., 107, 403
 Chateaufneuf, M., 838
 Chatterjee, K., 781, 783
 Chatterjee, S., 417, 754
 Chaturapruek, S., 1020, 1026
 Chavas, J., 890, 893
 Chavira, M., 906, 909
 Chayes, J. T., 41, 224, 441, 877
 Cheeseman, P., 40, 213, 216, 222
 Chellas, B. F., 1225–1229, 1234
 Chen, H., 468, 470, 1210
 Chen, J., 414, 417, 696, 703, 706, 843, 844, 1030
 Chen, J.-C., 81
 Chen, K., 843
 Chen, W. Y. C., 560
 Chen, X., 765
 Chen, Y., 937
 Chen, Y.-A., 32
 Cheng, K.-T. T., 1063
 Cherian, J., 44
 Cheswick, B., 530
 Chew, L., 415, 418, 1180, 1206–1214
 Chiarandini, M., 483
 Chickering, D., 474
 Chikh, M. A., 938
 Chistikov, D., 1039
 Chlamtáč, E., 311
 Chmelik, M., 781, 783
 Cho, J., 961
 Choi, A., 906, 909, 1015
 Choo, D., 418
 Chrzanowska-Jeske, M., 561
 Chtcherba, A., 41
 Chu, G., 968
 Church, A. S., 15

- Chvátal, V., 22–24, 36, 38, 40, 224, 257, 282, 285, 438, 441–443, 456, 1118
- Ciesielski, M., 1362
- Cimatti, A., 31, 32, 133, 157, 159, 739, 740, 742, 744, 745, 747, 748, 755, 756, 765, 779, 1172, 1224, 1248, 1274, 1285, 1286, 1290–1293, 1302, 1303, 1307–1309, 1313–1315
- Claessen, K., 134, 755, 845–847, 849, 850
- Clark, D., 93
- Clarke, E. M., 32, 108, 133, 134, 157–159, 739, 740, 742, 744, 745, 747–749, 753, 755, 756, 765, 799, 802, 804, 805, 808, 1137, 1172, 1193, 1194, 1196, 1198, 1255
- Clegg, M., 50, 222, 260, 272, 273, 275, 281, 1345, 1359
- Clote, P., 303
- Clymo, J., 1210
- Coarfa, C., 456
- Cocco, S., 451, 455, 869–871, 885, 888–890
- Codish, M., 81, 134, 156, 160, 417
- Cohen, B., 42, 213, 216–218, 464, 881, 1004, 1352
- Cohen, D., 564, 721
- Cohen, D. M., 835, 836
- Cohen, E., 936
- Cohen, G., 677, 836
- Cohen, J. D., 561
- Coja-Oghlan, A., 443, 445, 446, 451, 894
- Colange, M., 322
- Colbourn, C. J., 819, 821, 823–825, 827, 828, 830, 831, 836–838, 840, 841, 843
- Collautti, M., 492–494
- Colón, M. A., 805
- Conchon, S., 1297, 1299, 1303
- Condon, A., 1349
- Condrat, C., 418, 1359
- Conforti, M., 28
- Cook, B., 755, 797, 809–812, 814, 1159, 1281
- Cook, S. A., 21, 40, 121, 133, 222, 233, 234, 272, 305, 313, 316, 318, 322, 392, 439, 639, 1075, 1214
- Cook, W., 282, 285, 293, 302, 644
- Coppersmith, D., 44, 441
- Cordeiro, L. C., 937
- Cormen, T. H., 670
- Corneil, D. G., 714
- Cornuéjols, G., 28, 44, 1117
- Coste-Marquis, S., 54, 1140, 1168
- Cotton, S., 657, 1290, 1291, 1302
- Coudert, O., 31, 32, 1102, 1113–1115
- Coullard, C. R., 282, 285, 293, 302, 644
- Courcelle, B., 715, 718, 720
- Cousot, P., 804
- Cousot, R., 804
- Cover, T. M., 861, 862, 864
- Cox, D., 483
- Craig, W., 750, 814, 1304
- Crama, Y., 29, 50, 704, 706, 708
- Crato, N., 126, 140, 147, 160, 161, 439, 465–467, 473, 475
- Crawford, J. M., 80, 90, 92, 93, 151, 190, 214, 222, 416, 562, 563, 565, 849, 850, 1058, 1342
- Creignou, N., 41, 51, 673, 1139
- Croes, G. A., 42
- Crovella, M. E., 469
- Crowder, H., 965
- Cruz Da Silva, H., 937
- Cruz-Filipe, L., 154, 635
- Cugliandolo, L., 878
- Culberson, J. C., 257, 277
- Cull, P., 356
- Cunningham, W. H., 44
- Currie, D. W., 802
- Cygan, M., 694, 695
- Cyrluk, D., 797, 1274, 1297
- Czarnecki, K., 150, 162, 269, 402
- Dacek, S. T., 938
- Dadush, D., 305, 306
- D'Agostino, M., 407, 915, 1047, 1048, 1052, 1224, 1237, 1240

- Dagum, P., 1032, 1034
 Dala, S. R., 835, 836
 Dalal, M., 603
 Dalmao, S., 400, 717, 724, 993, 994,
 999, 1335, 1344, 1357
 Dalmau, V., 51, 262, 999, 1139
 D'Almeida, D., 917
 Damiano, R., 31, 371
 Danks, D., 937
 Dantchev, S. S., 257, 260, 271, 698,
 704, 726
 Dantsin, E., 26, 27, 43, 214, 674,
 676–680, 686
 Dantzig, G. B., 1114
 Danzig, G. B., 965
 Darbari, A., 154
 Darga, P. T., 530, 531, 534, 548, 549,
 552, 565
 Darras, S., 908
 Darwiche, A., 31, 107, 109, 116, 134,
 147, 148, 160–162, 247, 252–
 255, 267, 392, 401, 406, 477,
 906, 909, 993, 996, 998, 1001,
 1015
 Das, S., 811, 812
 Dasypodius, C., 13
 Daudé, H., 41, 870, 873
 Davenport, J. H., 1274
 Davies, J., 134, 158, 781, 783, 961–
 965, 1001, 1198
 Davis, E., 779, 781
 Davis, M., 19, 20, 75, 104, 111, 112,
 136, 138, 139, 141, 185, 238,
 240, 241, 392, 394–396, 401,
 644, 671, 674, 1102, 1103,
 1170, 1236–1238, 1240, 1242,
 1338
 Davis, R., 119
 Davis, W. T., 1066
 Davydov, G., 574, 618
 Davydova, I., 574, 618
 Dawande, M., 1117
 Dawson, K. A., 878
 de Colnet, A., 1036–1038
 De Faria Degelo, R., 937
 de Haan, R., 726
 de Lima, T., 134, 158
 De Loera, J. A., 275, 281
 de Oliveira, R. T., 937
 de Rezende, S. F., 258, 266, 268, 272,
 274, 281, 306, 307, 726
 Dechter, R., 84, 92, 105, 107, 118,
 119, 161, 213, 382, 945, 1005,
 1334, 1349, 1359
 del Val, A., 395, 404
 Delgrande, J. P., 1159
 Delmas, R., 937
 Dembo, A., 868, 877
 Demirovic, E., 936, 968
 Demopoulos, D. D., 456
 Demsky, B., 1039
 Denecker, M., 322
 Dequen, G., 193, 194, 472, 827, 908
 Deroulers, C., 887
 Dershowitz, N., 116, 124, 748, 755,
 1159, 1313
 Deters, M., 1315
 Detlefs, D., 810, 1282, 1285, 1286,
 1311, 1312
 Devadas, S., 1113, 1115
 Devendeville, L., 213, 908
 Devkar, A., 474, 490, 968
 Devriendt, J., 299, 311, 322, 401
 Didier, F., 957
 Dietmeyer, D. L., 561
 Dijk, van, T., 403, 418, 750
 Dilkina, B. N., 471, 712
 Dill, D. L., 32, 108, 745, 797, 811,
 1274, 1285, 1286, 1290, 1291,
 1297, 1301, 1303, 1307, 1308
 Dillig, I., 938
 Dimakis, A. G., 1020, 1023, 1025,
 1026, 1036
 Dimitrova, R., 936, 1039
 Dimopoulos, Y., 765, 770, 771
 Disch, S., 756
 Dixon, H. E., 287, 297, 1362
 Djamégni, C. T., 322
 Do, M. B., 765
 Dodaro, C., 134, 156–158, 954, 955,
 957, 959
 Doggett, A. J., 81
 Domhan, T., 483
 Domshlak, C., 1015

- Dongmin, L., 575
 Donini, F. M., 1172, 1256, 1257
 Dowling, W. F., 28, 672, 1058
 Downey, P. J., 1273
 Downey, R. G., 694–698, 704
 Drange, P. I. G. n. s., 719
 Dransfield, M. R., 31, 107, 383, 418,
 626, 833–835, 850
 Drechsler, R., 151, 1061, 1062, 1072,
 1077
 Dregi, M. S., 719
 Dreyer, A., 281, 418
 Droste, S., 214
 D'Silva, V., 792
 Du, X., 750
 Dubois-Lacoste, J., 488
 Dubois, O., 40, 189, 193, 194, 224,
 257, 377, 440, 472, 827, 864,
 869–871
 Dudek, J., 1038
 Duenas-Osorio, L., 1015, 1039
 Dufour, M., 191, 197, 201, 203, 1030
 Dunham, B., 396
 Duong, T.-T., 486
 Dutertre, B., 798, 1273, 1285, 1308,
 1309, 1313

 Ebbinghaus, H. D., 1269, 1272
 Eberhardt, F., 937
 Echenim, M., 1302
 Edmonds, J., 50, 222, 260, 272, 273,
 281, 1345, 1359
 Edwards, C. R., 561
 Eén, N., 50, 82, 116, 126, 134, 147,
 150, 153, 155–157, 161, 162,
 287, 385, 392, 393, 398, 399,
 401–403, 407, 411, 412, 417,
 418, 489, 635, 644, 645, 653,
 660, 661, 753, 756, 942, 944,
 947, 1009, 1051, 1118–1120,
 1242
 Eggersperger, K., 483, 485, 498, 499
 Eggersglüß, S., 1072, 1077
 Egly, U., 410, 1135, 1159, 1160, 1172,
 1181, 1183, 1189–1191, 1198,
 1199, 1204, 1206, 1210, 1213,
 1214

 Egyed, A., 134, 159, 660
 Ehlers, R., 403, 418, 938
 Ehlers, T., 720
 Eiben, A. E., 214
 Eichelberger, E. B., 1067
 Eijk, van, C. A. J., 755
 Eiter, T., 1159
 Ekin, O., 704, 706
 El Maftouhi, A., 40
 Elffers, J., 134, 160, 253, 255, 256,
 268, 269, 277, 287, 293–299,
 302, 307–311, 319
 Elgaard, J., 1274
 Emerson, A., 739, 740
 Emerson, E. A., 740
 Enderton, H. B., 1269
 Enescu, M. A., 1015
 Engelfriet, J., 720
 Engler, D. R., 797, 803
 Eppstein, D., 359
 Epstein, S., 498
 Erdős, P., 833, 834
 Erenrich, J., 1004
 Erez, A., 156
 Ermon, S., 159, 1016, 1020, 1023,
 1025, 1026, 1036, 1037
 Ernst, M., 79, 92, 769
 Esteban, J. L., 262, 263, 319, 1210
 Etherington, D. W., 603
 Evans, T., 829
 Even, S., 27, 587, 601

 Fagin, R., 1228, 1248
 Falkowski, B. L., 561
 Fang, Z., 908, 937
 Farahani, H. S., 930
 Faure, G., 635
 Fawcett, C., 483, 498
 Fazekas, K., 153, 392, 394, 400, 401,
 407, 409, 413, 1205
 Feder, T., 1139
 Feige, U., 45, 443, 451, 894
 Feigenbaum, J., 1349
 Fekete, Y., 417
 Feldman, R. E., 466
 Fellner, A., 658
 Fellows, M. R., 576, 694–698, 704,
 712, 726

- Feng, X., 937
Feng, Y., 938
Feo, T. A., 213
Fernàndez, C., 84, 87, 92, 468
Fernandez de la Vega, W., 40, 441, 877
Ferraris, P., 1158
Ferreira Jr., V., 213, 218, 219
Fethallah, H., 938
Feurer, M., 483
Fey, G., 1072, 1077
Fichte, J. K., 134, 160, 161, 252, 253, 255, 267, 720, 726
Filliâtre, J.-C., 1285
Filmus, Y., 262, 279
Fink, R., 1018
Finkel, A., 809
Finkelstein, L., 559, 562
Fischer, B., 154
Fischer, E., 710, 716, 720, 721, 994
Fischer, M. J., 233, 688
Fischmeister, S., 722
Fitting, M., 1223–1229, 1234–1237, 1256
Flahive, M., 356, 1282, 1283
Flanagan, C., 803, 1285, 1286, 1292
Fleischner, H., 572, 573, 725
Fleming, N., 305, 306, 311
Fleury, M., 394, 400, 401, 407, 409, 413
Flögel, A., 51, 54, 1146, 1147, 1149–1151, 1159, 1164, 1179, 1180, 1184, 1209, 1213, 1360
Flood, M. M., 42
Flum, J., 694, 695, 697, 698, 1269, 1272
Fogel, L. J., 42
Fomin, F. V., 373, 694, 695, 719, 722
Fontaine, P., 657, 1306
Fortet, R., 1091
Fortnow, L., 687
Fraise, H., 32
Fraleigh, J. B., 516, 517, 519
Franco, J., 24, 28–31, 33, 34, 36, 37, 41, 107, 213, 222, 224, 380, 383, 437, 438, 452, 608, 626, 724, 834, 835, 864, 883
Franco, J. V., 395, 418
Franconi, E., 1235, 1236
Frank, J., 93, 213, 216, 218
Franz, S., 869
Franzén, A., 1274, 1302, 1307–1309
Fränzle, M., 756
Fréchette, A., 495, 500
Fredman, M. L., 835, 836
Freeman, J. W., 24, 189, 192, 197, 372, 378, 395, 1342
Freeman, W. T., 876, 890, 1009
Fremont, D. J., 723, 1036–1039
Freudenthal, E., 1336
Freuder, E., 44, 498, 907
Frey, B. J., 382, 874, 876, 890
Fridshal, R., 396
Fried, D., 1038, 1039
Friedgut, E., 40, 224, 439, 864, 1336
Friedman, A. D., 1065, 1066
Friedman, J., 443
Frieze, A. M., 37, 38, 40, 224, 440, 447, 453, 454, 456, 867, 884, 888
Frisch, A. M., 80, 81, 92, 93, 156, 745, 944
Fröhlich, A., 148, 247, 417
Frohm, E. A., 32
Frost, D., 466
Fu, Y., 860
Fu, Z., 157, 417, 948, 949
Fuhrmann, O., 657
Fujita, H., 81, 134, 156, 820, 944, 947
Fujita, M., 133, 157, 159, 820, 822, 826, 827, 829, 831, 849
Fujiwara, H., 161, 1065, 1068, 1076, 1077
Fulkerson, D. R., 965
Furman, J., 43, 44, 907, 910
Furst, M. L., 766, 769, 770, 775, 777, 1211
Furtlehner, C., 890, 893
Gabàs, J., 906, 957, 965, 968
Gabbay, D. M., 1047
Galesi, N., 45, 258, 261–263, 275, 279–281, 304, 306, 321, 617, 726, 1210

- Galil, Z., 22
 Gallier, J. H., 28, 672, 1058
 Galvão Dos Santos Junior, E., 937
 Gamarnik, D., 44, 441
 Ganai, M. K., 151, 416, 739, 756, 803, 1051, 1056, 1058, 1063, 1292
 Ganesh, V., 150, 162, 269, 402, 722, 797, 1274, 1315
 Ganian, R., 718, 721–724
 Ganz, A., 1064, 1071
 Ganzinger, H., 134, 160, 1179, 1285, 1286, 1290, 1291, 1297, 1299
 Gaona, C. M., 32
 Garcia, M., 918, 936
 Gardner, M., 10, 11, 843
 Garey, M. R., 527, 696, 716, 995
 Garg, A., 305
 Gaschnig, J., 118, 161
 Gaspers, S., 697, 707–710, 721, 722
 Gathen, von zur, J., 1279
 Gavanelli, M., 80
 Ge, C., 831, 850
 Ge, Y., 1311, 1312
 Gebhardt, K., 419
 Gebser, M., 160, 287, 487
 Geffner, H., 781, 785, 906, 909
 Geffner, T., 781, 785
 Gelatt Jr., C. D., 42, 218, 1004
 Gelernter, H., 19
 Genesereth, M. R., 119
 Gennari, R., 1223, 1224
 Gent, I. P., 42, 79, 81, 86, 88, 92, 93, 134, 150, 155–157, 213, 219, 465, 564, 820, 841, 850, 1159, 1168, 1342, 1343, 1352, 1355, 1358
 Gentzen, G., 316, 336
 Gerevini, A., 483, 769
 German, S., 1256, 1268, 1278
 Gerschenfeld, A., 454
 Gershman, R., 393, 396, 404, 406, 1313
 Ghasemi, M., 936
 Ghasemzadeh, M., 1344
 Ghassem-Sani, G., 781
 Ghilardi, S., 1302, 1303, 1305
 Giannoros, P. A., 944
 Gilbert, J. R., 264, 265
 Gilmore, P., 19
 Ginsberg, M. L., 92, 151, 161, 213, 287, 297, 562, 563, 565, 849, 1362
 Giovanardi, A., 755, 1159, 1162, 1168, 1184–1186, 1199, 1200, 1338, 1342
 Giovanardi, M., 1168
 Giráldez-Cru, J., 134, 160, 268, 269, 293, 296–299, 307, 308, 310, 722
 Girvan, M., 722, 970
 Giunchiglia, E., 31, 76, 1062, 1158–1160, 1164, 1166–1168, 1172, 1179, 1180, 1183, 1184, 1189, 1198, 1199, 1224, 1237, 1239, 1242, 1244, 1247, 1248, 1268, 1285, 1286, 1290, 1292, 1343, 1355, 1358, 1360
 Giunchiglia, F., 1172, 1223, 1224, 1234, 1235, 1237, 1239, 1240, 1242–1244, 1248, 1268, 1286, 1290, 1292
 Giunchiglia, P., 31
 Givry, de, S., 43, 906, 911, 913, 930
 Glaß, M., 1092
 Glass, T., 417
 Gleixner, A., 299
 Glowatz, A., 1072, 1077
 Gocht, S., 268, 269, 293, 296–299, 307, 308, 401, 414, 416, 1032
 Godefroid, P., 770
 Goel, A., 1279, 1303, 1306
 Goel, P., 1062, 1065, 1068, 1076
 Goemans, M. X., 45, 48, 917, 940, 965
 Goerdts, A., 26, 38, 40, 43, 214, 318, 441, 443, 658, 676–679, 686
 Gogate, V., 382, 1005, 1018
 Goldberg, A., 33, 222, 437
 Goldberg, E. I., 116, 128, 148, 154, 162, 244, 249, 385, 635, 642, 653, 658, 1242
 Goldreich, O., 1016, 1029
 Goldsmith, J., 1334, 1335

- Gomes, C. P., 126, 134, 140, 147, 148, 151, 159–161, 224, 439, 465–468, 470–476, 490, 704, 707, 712, 822, 823, 844, 917, 1006–1008, 1016, 1020, 1023, 1025, 1026, 1036, 1037, 1159, 1172, 1242
- Gomory, R. E., 282, 1096, 1116, 1117
- Göös, M., 272, 281, 303, 305, 306
- Gordon, U., 216, 224, 890, 892
- Gottlieb, J., 214
- Gottlob, G., 714, 715, 721, 722, 994
- Goultiaeva, A., 1061, 1198, 1205, 1206
- Govindan, R., 530
- Graça, A., 917, 933, 938
- Graf, S., 805
- Graham, D., 489
- Graham, R., 832
- Gramm, J., 43, 44
- Grandoni, F., 373
- Granmo, O.-C., 214
- Grant, S. A., 465
- Gray, R. M., 1025
- Greenbaum, S., 86, 417, 1047, 1160, 1193
- Grégoire, B., 635
- Grégoire, É., 75, 134, 158, 159, 213, 219, 399, 417, 585, 917
- Gregorio, P. D., 878
- Gretton, C., 769
- Greuel, G.-M., 281
- Gribomont, E. P., 1306
- Griggio, A., 1274, 1302, 1307–1309, 1313–1315
- Grigore, R., 157, 938, 1200
- Grigoriev, D., 45, 271, 276, 311
- Grinten, van der, A., 392
- Groce, A., 808
- Grochow, J. A., 311
- Grohe, M., 694, 695, 697, 698, 721
- Groote, J. F., 31, 111, 203, 312, 417
- Grötschel, M., 45, 49
- Grumberg, O., 739, 745, 756, 802, 804, 805, 1194, 1255
- Grundy, J., 1303, 1306
- Gu, J., 42, 213, 380
- Guerra e Silva, L., 151
- Guerra, J., 917, 938
- Gulwani, S., 1315
- Guo, J., 697
- Gupta, A., 740, 753, 755, 756, 803, 814, 1058, 1061, 1292
- Gupta, S., 1065, 1067, 1077
- Gurfinkel, A., 636
- Gutin, G., 703
- Gyögy, A., 489
- Gyssens, M., 721
- Haarslev, V., 1224, 1237, 1242
- Habet, D., 213, 906, 908, 910, 912, 913, 918, 919
- Hachtel, G. D., 32, 561
- Hadarean, L., 1285
- Hadzic, V., 1191
- Hagen, G., 1285, 1286, 1290, 1291
- Hague, M., 938
- Hahn, G. J., 491
- Hähnle, R., 915, 1047
- Hájek, P., 322
- Hajiaghayi, M. T., 24, 37, 44, 224, 440, 441, 453
- Haken, A., 22, 222, 256, 305, 313, 318, 319, 321, 392, 639, 644, 1099
- Hakoniemi, T., 271
- Halkes, G., 823
- Halperin, E., 46
- Halpern, J. Y., 1223–1226, 1228, 1247, 1248, 1256, 1257
- Hamadi, Y., 143, 393, 399, 408, 419
- Hamaguchi, K., 745, 1255
- Hameurlain, N., 938
- Hammer, P. L., 29, 49, 50, 704, 706, 708, 1087, 1090, 1093
- Hammoudeh, Z., 1020, 1025, 1026, 1028
- Han, C., 414, 416
- Han, C.-S., 1030, 1031
- Han, H., 254, 393, 399, 402, 406, 408, 409, 655
- Hanatani, Y., 93
- Hanna, Z., 116, 124, 392, 748, 755, 1063, 1159, 1274, 1302, 1313

- Hansen, P., 30, 42, 43, 50
 Hao, J.-K., 214
 Hapke, F., 1072, 1077
 Harandi, M. T., 1303–1305
 Harchol-Balter, M., 469
 Hardy, G. H., 360
 Harrison, J., 109, 110
 Harrison, M. A., 523, 560
 Hart, J. W., 1172
 Hartley, S. J., 840
 Hartman, A., 836, 838
 Hartmann, A. K., 879, 881, 882, 895
 Hasegawa, R., 81, 134, 156, 820, 944, 947
 Haslum, P., 781
 Håstad, J., 45, 263, 320, 382, 917, 1211
 Haubelt, C., 1092
 Haven, G. N., 24, 36
 Hayes, P. J., 21
 Hebrard, E., 92
 Heguiabehere, J., 1224
 Heintz, J., 1274
 Heisinger, M., 394, 400, 401, 407, 409, 413
 Heizmann, M., 938
 Heljanko, K., 409, 419, 740, 741, 745–747, 750, 770–772, 774, 775, 779
 Helmberg, C., 45
 Helmert, M., 497
 Hemaspaandra, E., 51, 1139
 Henderson, M., 617
 Henftling, M., 1064, 1071
 Henkin, L. A., 17, 1145, 1172
 Henzinger, T. A., 770, 814
 Heras, F., 134, 157, 158, 660, 906, 908, 910–913, 918, 919, 930, 934, 939, 942, 961, 966, 968, 1124
 Herbstritt, M., 1159
 Herde, C., 756
 Herlinus, C., 13
 Hertel, P., 251
 Hertz, J., 861
 Herwig, P., 184, 834, 835
 Heuerding, A., 1223, 1235, 1236
 Heule, M. J. H., 47, 150, 151, 154, 159, 160, 162, 191, 192, 197, 198, 201, 203, 206, 244, 249, 251, 281, 313, 315, 322, 339, 356, 378, 380, 392–394, 398, 401, 403, 406–411, 413–418, 603, 635, 636, 640–645, 650–652, 655, 657, 661, 721, 834, 835, 966, 970, 1030, 1057, 1179, 1199–1201, 1203, 1205, 1213
 Heuser, A., 1015
 Hinde, L., 1209–1214
 Hiriart-Urruty, J.-B., 361
 Hirsch, E. A., 26, 43–45, 213, 214, 218, 235, 271, 311, 487, 602, 674, 676–682, 686
 Hlinený, P., 721
 Hnich, B., 838, 839
 Ho, Y. C., 33
 Hochbaum, D., 1281
 Hoesel, van, S. P. M., 719
 Hoeve, van, W.-J., 917
 Hoey, J., 1348
 Hofer, D., 1362
 Hoffmann, J., 251, 254, 471, 1006, 1008, 1015, 1016, 1025, 1026
 Hogg, T., 465, 466
 Hojati, R., 108, 109
 Holland, J. H., 42
 Hölldobler, S., 80, 944, 945
 Hollunder, B., 1223, 1235, 1236
 Holte, R. C., 42, 213
 Holzmann, G., 740
 Hong, C.-D., 938
 Hong, Y., 32
 Honkala, I., 677
 Hooker, J. N., 28, 214, 285, 308, 374, 377, 392, 1110, 1116, 1342, 1361, 1362
 Hoory, S., 45, 257, 617, 657
 Hoos, H. H., 42, 43, 85, 87, 92, 159, 213, 217–219, 474, 483–488, 490–500, 968, 1168, 1352, 1358
 Hopcroft, J. E., 263, 527, 552, 725
 Horiyama, T., 93

- Horrocks, I., 1223, 1224, 1237, 1239,
 1240, 1242, 1243, 1247, 1291
 Horvitz, E., 474
 Hristov, N., 1352, 1358
 Hrubeš, P., 305, 318
 Hsiang, H., 827, 831
 Hsiao, M. S., 1360
 Hsieh, E. R., 1066
 Hsu, E. I., 383
 Hsu, T., 1191
 Hu, A., 483, 484, 499
 Hu, A. J., 802, 1348
 Hu, G., 208
 Huang, C.-Y. R., 1059, 1063
 Huang, G., 936
 Huang, J., 31, 107, 109, 116, 126,
 134, 148, 160, 247, 313, 1018,
 1313
 Huang, K., 866
 Huang, P., 831, 850
 Huang, W., 43, 938, 968
 Huber, M., 1032
 Huertas, A., 916
 Hulgaard, H., 1051
 Hunt, Jr., W. A., 154, 249, 251, 313,
 315, 322, 411, 635, 645, 650,
 652, 657, 1179
 Hurst, S. L., 561
 Husfeldt, T., 716
 Hustadt, U., 1223, 1224
 Hutter, F., 42, 159, 213, 218, 219,
 483–487, 490–495, 497–499,
 968
 Huynh, T., 262, 279, 300, 301, 306
 Hwang, J., 384
 Hwang, L. J., 32, 108, 745
 Hyttinen, A., 937, 963–965
 Hyvärinen, A. E. J., 396
 Ignatiev, A., 134, 151, 155, 156, 158–
 160, 163, 323, 914, 918, 935–
 937, 961, 963, 1015, 1039
 Immerman, N., 673
 Impagliazzo, R., 23, 50, 222, 233,
 239, 257, 258, 260, 261, 265,
 266, 268, 270, 272, 273, 275,
 276, 280, 281, 306, 319–321,
 330, 442, 680, 681, 683–686,
 697, 999, 1210, 1344, 1345,
 1359
 Inoue, K., 158, 402
 Interian, Y., 471
 Irving, R. W., 88
 Iser, M., 148, 392, 417
 Ishtaiwi, A., 43, 213, 218
 Ismail, H. I., 937
 Istrate, G., 41
 Itai, A., 27, 28, 587, 601
 Itsykson, D., 235, 271, 312
 Ivancic, F., 803, 814
 Iverson, V., 1061
 Ivrii, A., 158, 1020, 1025, 1026, 1039
 Iwama, K., 27, 34, 79, 84, 92, 93,
 213, 218, 576, 577, 679, 711
 Iwanuma, K., 402
 Izquierdo, I., 937
 Izza, Y., 134, 159
 Jabbour, S., 134, 143, 155, 156, 159,
 399, 564, 917, 938
 Jackson, D., 157, 797
 Jackson, P., 1047, 1160
 Jacob, J., 32
 Jaffar, J., 1281
 Jager, G., 1235, 1236
 Jahanjou, H., 688
 Jain, H., 808, 814
 Janicic, P., 1285
 Janota, M., 134, 152, 157, 158, 160,
 417, 936, 937, 968, 970, 971,
 1177, 1191–1193, 1195, 1196,
 1198, 1200, 1204–1213
 Jansen, T., 214
 Janson, S., 224, 863, 874
 Järvisalo, M., 134, 150, 151, 154, 158,
 159, 162, 244, 251, 267, 268,
 312, 392–394, 396, 398, 406–
 411, 413, 417, 487, 641, 642,
 644, 720, 930, 937, 940, 961,
 963–967, 1057, 1064, 1199,
 1201
 Jaumard, B., 30, 42, 43, 50
 Jeavons, P., 80, 564, 721
 Jefferson, C., 564

- Jégou, P., 564
 Jensen, T. R., 613
 Jeroslow, R. G., 24, 377, 380, 1117, 1342
 Jerrum, M. R., 1004, 1016, 1018, 1019, 1039
 Jevons, W. S., 11
 Jha, A. K., 718
 Jha, N. K., 1065, 1067, 1077
 Jha, S., 756, 805, 808, 1194, 1285, 1315
 Jhala, R., 814
 Ji, L., 840
 Jia, H., 467, 468
 Jia, W., 696
 Jiang, H., 908, 937
 Jiang, J.-H. R., 414, 416, 1030, 1031, 1039, 1181, 1182, 1191, 1204, 1205, 1210, 1211, 1213
 Jiang, P., 1036
 Jiang, R., 417
 Jie, Z., 968
 Jin, H. S., 31, 107, 406
 Johannsen, J., 240, 251, 253–255, 263, 1210
 Johansson, T., 836
 Johnson, D. R., 696, 716
 Johnson, D. S., 45, 215, 380, 381, 527, 703, 725, 917, 995
 Johnson, E. L., 965
 Johnson, S. M., 965
 Johnston, M. D., 42, 215
 Jones, B., 1032
 Jones, D. R., 486
 Jonsson, P., 781
 Jordan, C., 1177
 Jordan, M. I., 890, 1009, 1361
 Jose, M., 938
 Joshi, R., 1285, 1286, 1292
 Joshi, S., 134, 158, 944, 945, 952, 953, 955, 968, 1038
 Jovanovic, D., 287, 293, 1285
 Jukna, S., 1210
 Juniwal, G., 1315
 Junker, U., 160
 Junttila, T. A., 151, 396, 414, 548, 565, 740, 741, 745–747, 750, 1030, 1031, 1052, 1063, 1064, 1285, 1286, 1290–1292, 1302, 1303, 1307–1309
 Jurkowiak, B., 559
 Jussila, T., 51, 740, 748, 755, 1159
 Kabanets, V., 685
 Kadioglu, S., 491, 492, 494, 496, 497, 968
 Kaiser, A., 660
 Kalech, M., 160
 Kalla, P., 418, 1359, 1362
 Kam, T., 1102, 1115
 Kamath, A., 40, 214, 1336
 Kamath, P., 272, 303, 305
 Kambhampati, S., 765
 Kaminski, R., 160, 287, 497
 Kanefsky, B., 40
 Kanj, I. A., 696, 706, 726
 Kannan, R., 26, 43, 214, 676–679, 686, 1279
 Kapoor, A., 28
 Kaporis, A. C., 24, 37, 224, 440, 453
 Kapur, D., 1314, 1315
 Karamcheti, V., 1336
 Karloff, H., 46, 917
 Karmarkar, N., 214, 1273
 Karp, R. M., 38, 49, 222, 443, 456, 725, 889, 961, 996, 1003, 1017, 1019, 1032–1034, 1114, 1345
 Karpinski, M., 51, 54, 134, 156, 1146, 1147, 1149–1151, 1159, 1164, 1179, 1180, 1184, 1209, 1213, 1360
 Kasif, S., 79, 92
 Kask, K., 84, 92, 382
 Kaski, P., 548, 565, 716, 881
 Kaski, S., 937
 Katebi, H., 268, 548, 565
 Katsirelos, G., 134, 155, 158, 160, 313
 Katz, J., 748, 755, 1159
 Kauers, M., 282, 418, 1183, 1184
 Kaufmann, B., 160, 287, 487, 954
 Kaufmann, D., 282, 418
 Kaufmann, M., 154, 635, 645, 657
 Kautz, H. A., 23, 42, 76, 90, 93, 119, 121, 122, 127, 133, 134, 140,

- 141, 147, 148, 157, 159–161,
 213, 216, 218, 241, 251, 371,
 392, 400, 401, 439, 464, 467,
 468, 472–476, 646, 755, 765,
 766, 769, 770, 775, 777, 881,
 993, 996, 1000, 1004, 1048,
 1242, 1344, 1347, 1349, 1352,
 1357, 1362
 Kearns, M. J., 90, 190, 416, 1351
 Kégl, B., 483
 Keinänen, M., 745
 Keller, C., 635
 Kenefsky, B., 222
 Kern, W., 26, 596, 679, 686
 Kernighan, B. W., 42, 215
 Kersting, K., 1039
 Khanna, S., 51, 673, 1139
 Khasidashvili, Z., 392
 Khatri, S. P., 938
 KhudaBukhsh, A., 483, 487, 499
 Kiesl, B., 160, 251, 315, 339, 393,
 401, 410, 411, 413, 635, 636,
 640–643, 645, 1179, 1213
 Kilby, P., 151, 471, 1006
 Kim J. H., 41, 224, 392, 441, 703,
 724, 750, 821, 877, 1361
 Kirkpatrick, S., 40–42, 151, 194, 213,
 214, 216, 218, 222–224, 438,
 442, 465, 878, 885, 890, 892,
 1004
 Kirkpatrick, T. R., 869
 Kirousis, L. M., 24, 37, 40, 41, 224,
 440, 442, 453, 885, 1336
 Kitagawa, S., 80
 Kitchaev, D. A., 938
 Kitchen, N., 417
 Klarlund, N., 1274
 Kleene, S. C., 15
 Kleer, de, J., 119, 963, 1107
 Klein, A., 483
 Kleinberg, J. M., 26, 43, 214, 676–
 679, 686
 Kleinberg, R., 488, 489
 Kleine Büning, H., 24, 30, 51, 54,
 134, 1143–1147, 1149–1152,
 1154, 1159, 1164, 1171, 1360
 Klerk, de, E., 45, 46
 Klieber, W., 81, 134, 158, 1177, 1193,
 1196, 1198
 Kloks, T., 716, 719
 Klotz, V., 1344
 Knop, A., 312
 Knuth, D. E., 30, 134, 217, 356, 357,
 365, 398, 484
 Kochemazov, S., 151
 Kodandapani, K. L., 1066
 Koehler, J., 765, 770, 771
 Kohli, R., 158
 Koifman, M., 1313
 Koivisto, M., 716
 Kojevnikov, A., 213, 218, 271
 Kokkala, J. I., 249, 269
 Kolář, M., 890, 892
 Kolaitis, P. G., 311, 312, 683, 713,
 999, 1139
 Kolb, S., 1039
 Kolokolova, A., 306
 Konev, B., 154, 159, 636, 645
 Konolige, K., 213
 Kordon, F., 322
 Korhonen, J. H., 396
 Korhonen, T., 967
 Kornilowicz, A., 779, 1224, 1248, 1285,
 1286, 1290–1293, 1302
 Koroth, S., 305
 Korovin, K., 134, 160, 399
 Koshimura, M., 81, 134, 156, 820,
 937, 944, 947
 Koster, A. M. C. A., 719
 Kothari, P., 311
 Kotthoff, L., 495, 500
 Kouril, M., 31, 107, 383, 418, 626,
 834, 835
 Kovács, L., 134, 160
 Kowalik, L., 694, 695
 Kowalski, R., 21
 Kołodziejczyk, L., 254, 262, 280
 Krajíček, J., 235, 259, 270, 272, 312,
 317, 319, 320, 322, 750, 1210,
 1214
 Kranakis, E., 40, 41, 224, 440, 442,
 885, 1336
 Kratsch, D., 373, 719
 Krauth, W., 868

- Kravets, V. N., 562
 Kreher, D. L., 838
 Krishnamurthy, B., 261, 322, 562
 Krishnamurthy, S., 881
 Krishnamurti, R., 158
 Krishnan, S. C., 108, 109
 Krishnaswamy, S., 417
 Krivelevich, M., 38, 443, 894
 Krizanc, D., 40, 41, 224, 440, 442, 885, 1336
 Kroc, L., 224, 225, 1007–1009
 Kroening, D., 157, 159, 742, 745, 748, 750, 755, 792, 797, 798, 802, 808–811, 814, 1159, 1274, 1285, 1310, 1311, 1315
 Krogh, A., 861
 Krohm, F., 417, 739, 1051, 1056, 1058, 1063
 Krstić, S., 1303, 1306
 Krzakała, F., 224, 444, 446, 454, 869, 870, 872, 873, 877, 892, 895
 Kschischang, F. R., 382, 874, 876, 890
 Küchlin, W., 660
 Kudrle, J. M., 843
 Kuegel, A., 906, 908, 910
 Kuehlmann, A., 50, 151, 287, 290, 291, 296, 298, 416, 417, 739, 750, 754, 755, 1051, 1056, 1058, 1063, 1111, 1112, 1114
 Kügel, A., 918
 Kukula, J., 31, 371
 Kullmann, O., 29, 30, 92, 154, 158, 159, 190, 193, 197, 203, 205, 351, 355, 356, 362, 365, 367, 368, 371, 373, 374, 376, 409, 572–575, 578–580, 583, 585, 587, 589, 590, 594, 598–605, 607–610, 613, 614, 616–620, 622, 623, 625, 626, 636, 640, 642, 645, 655, 674, 681, 682, 725
 Kumar, P., 968
 Kumar, S. R., 822
 Kunz, W., 417, 797, 1069
 Kurosawa, K., 836
 Kurshan, R. P., 740, 750, 805
 Kurtz, T. G., 870
 Kushilevitz, E., 301, 306
 Kutzner, F., 417
 Kwon, G., 81
 Ladner, R., 1223, 1224, 1226, 1228
 Lagarde, G., 266, 280
 Lagergren, J., 930
 Lagniez, J.-M., 134, 153, 158, 159, 392, 417, 661, 1025
 Lagoudakis, M., 498
 Lahiri, S. K., 810, 812, 1159, 1274, 1275, 1279, 1281, 1284, 1290, 1307
 Laird, P., 42, 215
 Laitinen, T., 414, 1030, 1031
 Lalas, E. G., 24, 37, 224, 440, 453
 Lam, C. W. H., 825
 Lambalgen, van, M., 834, 835
 Lammich, P., 635
 Lang, J., 76, 105, 1025
 Lang, S., 591
 Lange, M., 745
 Lanka, A., 443
 Lardeux, F., 214
 Larochelle, H., 483
 Larrabee, T., 133, 161, 416, 1064, 1069
 Larrosa, J., 43, 906, 908, 910–913, 930, 939, 942
 Lasserre, J. B., 45, 46, 311
 Latvala, T., 740, 741, 745–747, 750
 Lau, D., 591
 Laurent, M., 45, 311
 Lauria, M., 253, 255, 256, 258, 260, 262, 271, 274, 275, 277, 279, 281, 302, 319, 321, 726, 1210
 Lavagno, L., 1069, 1074
 Lavine, S., 12
 Lawler, E. L., 1113
 Lawlor, A., 878
 Le Berre, D., 54, 127, 134, 158, 160, 286, 287, 298, 309–311, 395, 418, 487, 936, 946, 947, 1140, 1168
 Leahu, L., 917

- Lee, C.-C., 1059
Lee, C. Y., 31
Lee, J., 275, 281, 1061, 1062
Lee, N.-Z., 1039
Legay, A., 1015
Lei, Z., 968
Leino, K. R. M., 803
Leiserson, C. E., 670
Lemaréchal, C., 361
Lengauer, T., 265
Leone, M., 869, 895
Leone, N., 721
Lerda, F., 157, 159, 802
Letchford, A. N., 1117
Letombe, F., 54, 968, 1140, 1168
Lettmann, T., 54, 134, 582, 708, 1147,
1152, 1154
Letz, R., 1166, 1167, 1179, 1180, 1184
Levesque, H. J., 40, 42, 133, 213–
215, 218, 222, 438, 456, 860,
1336, 1352
Levin, L. A., 133, 233
Levitt, J. R., 797, 1274, 1301
Levy, J., 134, 158, 267, 722, 913, 914,
942, 950, 953, 960, 961
Lewis, C. I., 16
Lewis, H. R., 28, 707
Leyton-Brown, K., 159, 474, 483–491,
494–499, 968
Li, C. M., 43, 75, 82, 127, 150, 157,
158, 162, 190, 193, 199, 205,
206, 213, 374, 377, 378, 383,
402, 404, 408, 409, 414, 416,
471, 559, 906, 908, 910, 912,
915, 917–919, 930, 935, 937,
939, 967
Li, F., 311
Li, N., 937
Li, S., 722, 1313
Li, X. Y., 213
Li, Y., 408, 409
Li, Z., 159
Liang, J. H., 150, 162, 269, 402
Liang, T., 1315
Liao, S., 1113, 1115
Liao, X., 937
Liberatore, P., 105, 1172
Lichtenstein, D., 30
Lichtenstein, O., 742, 745
Lierler, Y., 483
Liew, V., 311
Liffiton, M. H., 134, 157–160, 534,
548, 552, 565, 585, 601, 660,
917, 937, 938, 963, 1026, 1124
Liggett, T. M., 879
Lillibridge, M., 803
Limaye, R., 1285
Lin, A. W., 938
Lin, B., 108
Lin, H., 906, 908
Lin, J., 968
Lin, P.-C. K., 938
Lin, S., 42, 215
Lin, T.-H., 1059
Lincoln, P., 1297
Lindauer, M., 485, 487, 493, 494, 497–
499
Ling, A. C., 1159
Linial, N., 30, 257, 572, 608, 609, 725
Lisitsa, A., 154, 159, 636, 645
Litsyn, S., 677
Littlewood, J. E., 360
Littman, M. L., 498, 781, 783, 993,
999, 1331–1336, 1338, 1339,
1341–1348, 1350, 1351, 1353,
1356
Liu, J., 1015
Liu, Y., 81, 134, 156, 944
Lobstein, A., 677
Lodha, N., 720
Lodi, A., 1117
Loeliger, H.-A., 382, 874, 876, 890
Logemann, G., 20, 75, 112, 136, 138,
139, 141, 185, 238, 241, 395,
674, 1236–1238, 1240, 1242,
1338
Loiacono, C., 750
Lokshtanov, D., 694, 695, 719, 722
Lonca, E., 310, 417, 418, 1025
Long, D. E., 804
Lonlac, J., 155
Lonsing, F., 393, 410, 413, 654, 1177,
1181, 1183, 1189–1191, 1198,
1199, 1201, 1204, 1206, 1210,

- 1213, 1214
 López-Ibáñez, M., 488
 Lopuszański, J., 1285, 1311, 1312
 Lotem, A., 765
 Lourenço, H. R., 484
 Lovász, L., 44, 45, 311, 725
 Loveland, D. W., 20, 75, 112, 136, 138, 139, 141, 185, 238, 241, 395, 674, 1236–1238, 1240, 1242, 1338
 Lozinskii, E. L., 996, 997
 Lu, F., 1063
 Lu, Y., 756, 805, 1194
 Lü, Z., 145, 150, 402, 408, 409
 Luby, M. G., 148, 453, 473, 996, 1003, 1017, 1019, 1032–1034
 Lucier, B., 488, 489
 Luckhardt, H., 190, 356, 362, 367, 373, 374, 587, 600, 602, 674, 681
 Łuczak, T., 863, 874
 Lueker, G. S., 719
 Lukaszewicz, M., 1092
 Luks, E. M., 92, 151, 562, 563, 565, 849, 1362
 Lund, C., 43, 1349
 Luo, C., 495, 938, 968
 Luo, M., 145, 150, 402, 408, 409
 Lusena, C., 1335
 Luu, L., 1039
 Lynce, I., 81, 92, 134, 147, 148, 158, 162, 268, 287, 395, 417, 474, 476, 477, 579, 585, 601, 823, 841, 850, 917, 918, 933, 935, 936, 938, 945, 952, 953, 955, 961, 969–971, 1056, 1104, 1313

 Ma, F., 830, 831, 850
 Ma, S. K., 866
 Maaren, van, H., 24, 29, 46, 47, 151, 191, 192, 197, 198, 201, 203, 206, 281, 356, 375, 378, 380, 414, 416, 585, 603, 609, 610, 834, 835, 1030
 Maciel, A., 321
 Macii, E., 32
 MacIntyre, E., 93
 Mackey, J., 636, 645, 655
 Madigan, C. F., 24, 25, 116, 118–121, 124, 126, 127, 133, 140, 142, 143, 147–149, 161, 162, 234, 241, 247, 254, 269, 385, 397, 398, 474, 477, 553, 562, 756, 797, 1000, 1058, 1065, 1077, 1104, 1109, 1110, 1118, 1167, 1179, 1242, 1286, 1288, 1292, 1352, 1355
 Madras, N., 996, 1004, 1019, 1032–1034
 Madre, J. C., 31, 32, 1113, 1115
 Magen, A., 45
 Magnanti, T., 1114
 Magnard, T., 253, 255
 Mahajan, M., 44, 703, 1210, 1213
 Mahajan, S., 45
 Maher, M. J., 1281
 Mahjoub, A. R., 49
 Mai, S. T., 81
 Maidl, M., 740
 Majercik, S. M., 783, 993, 999, 1331–1336, 1338, 1339, 1341–1348, 1350–1353, 1356–1358
 Majumdar, R., 808, 814, 938, 1039, 1314, 1315
 Makowsky, J. A., 28, 710, 715, 716, 720, 721, 994
 Maler, O., 1290, 1291, 1302
 Malik, A., 134, 159, 160
 Malik, S., 24, 25, 116, 118–121, 124–127, 133, 134, 140, 142, 143, 147–149, 154, 157, 158, 161, 162, 234, 241, 247, 254, 269, 385, 397, 398, 417, 474, 477, 553, 561, 562, 585, 635, 645, 653, 654, 754, 756, 797, 938, 948, 949, 1000, 1020, 1025, 1026, 1039, 1058, 1065, 1077, 1104, 1109, 1110, 1118, 1166, 1167, 1179–1181, 1184, 1189, 1240, 1242, 1285, 1286, 1288, 1290, 1292, 1313, 1352, 1355
 Maliotov, D., 937

- Malitsky, Y., 485, 491–494, 496, 497, 968
- Malkin, P. N., 281
- Malone, B., 930, 937
- Manandhar, S., 1353–1355, 1362
- Mandelbrot, B., 466
- Mandler, J., 40, 224, 257, 440, 869–871
- Maneva, E. N., 224, 225, 383, 890, 892
- Mangal, R., 938
- Mangassarian, H., 748, 755, 917, 937, 1159
- Manlove, D., 88
- Manna, Z., 755, 1273, 1305
- Manolios, P., 417, 1315
- Manquinho, V. M., 134, 151, 155–158, 287, 944, 945, 950, 952, 953, 955, 961, 969–971, 1116, 1118, 1123
- Mansour, Y., 1351
- Manthey, N., 134, 150, 157, 310, 313, 392, 403, 418, 419, 487, 499, 642, 644, 650, 944, 945, 967
- Mantovani, J., 803
- Manyà, F., 44, 80, 84, 87, 92, 134, 145, 150, 156–158, 267, 402, 408, 409, 906, 914, 919, 930, 935–937, 939, 942, 967
- Maratea, M., 1062, 1285, 1290
- Marben, J., 498
- Marcel Kevin, T., 936
- Marchiori, E., 214
- Marek-Sadowska, M., 561
- Marek, V. W., 154, 159, 590, 594, 598, 599, 603, 604, 607, 616, 622, 636, 645, 655, 833–835, 850
- Margulies, S., 275, 281
- Maric, F., 1285
- Marin, P., 1198, 1199
- Marinescu, M.-C., 134, 158
- Markov, I. L., 31, 43, 50, 107, 109, 417, 418, 530–532, 534, 546, 548, 549, 552, 563–565, 585, 644, 849, 947, 1313
- Markström, K., 277
- Marques-Silva, J. P., 24, 25, 81, 116, 118–121, 124, 126, 133, 134, 137, 140, 142–144, 147–149, 151, 152, 154–163, 268, 287, 323, 385, 395, 417, 473, 474, 476, 477, 553, 579, 585, 601, 636, 660, 755, 918, 919, 933–938, 942, 950, 952–955, 961, 963, 966–968, 1015, 1039, 1056, 1064, 1065, 1104, 1108–1110, 1116, 1118, 1123, 1124, 1167, 1179, 1191–1193, 1198, 1200, 1210, 1212, 1242, 1313
- Marquis, P., 54, 76, 105, 298, 417, 1001, 1025, 1140
- Martin, B., 271, 698, 704, 726
- Martin, J. N., 7
- Martin, M. M. K., 1315
- Martin, O., 484
- Martins, R., 134, 158, 287, 930, 936, 938, 944, 945, 952, 953, 955, 968–971
- Marx, D., 694, 695, 701, 721
- Massacci, F., 1030, 1048, 1172, 1223, 1224, 1228, 1235–1237, 1256, 1257
- Massarotto, A., 1062
- Matheis, O., 954
- Mathew, M., 150, 162
- Matiyasevich, Y. V., 1274
- Matos, P. J., 968
- Matsliah, A., 267, 268
- Mayer-Eichberger, V., 134, 157
- Mazure, B., 75, 134, 158, 213, 219, 399, 417, 585
- Mazzola, J. B., 1091
- McAllester, D. A., 42, 76, 90, 93, 127, 136, 213, 395, 769, 881, 1048, 1102, 1103, 1347, 1352
- McAloon, K., 467
- McBride, R., 299
- McCluskey, E. I., Jr., 19, 158
- McCreesh, C., 299
- McCuaig, W., 579, 613, 619, 622
- McCune, W., 820, 831
- McDonald, C. B., 32
- McDonald, I., 86, 88

- McFadden, J., 490, 496
McGeer, P. C., 133
McGeoch, L. A., 215
McIlraith, S. A., 383, 936
McKay, B. D., 526, 548, 552, 563, 565
McKenzie, P., 263
McMillan, K. L., 32, 108, 159, 636, 657, 660, 740, 745, 750, 751, 753, 754, 756, 814, 1313–1315
Meel, K. S., 158, 159, 401, 414, 416, 418, 937, 1015–1020, 1023, 1025–1028, 1030–1032, 1034–1039
Megiddo, N., 1281
Mehta, D., 492–494
Meinel, C., 1344
Meir, O., 272, 274, 281, 307
Meltzer, B., 16
Memisevic, R., 1358
Menard, S. B., 843
Mencía, C., 134, 158–160, 968
Mendelsohn, E., 836
Mengel, S., 298
Mertens, S., 222, 444, 872, 873
Mertz, I., 251, 267, 305
Merz, S., 657
Meseguer, P., 43, 911
Metin, H., 322
Metodi, A., 160, 417
Metropolis, N., 1004
Meyer, A. R., 51, 688, 781, 1137, 1138
Mézard, M., 40, 214, 222, 224, 383, 454, 870, 872, 873, 890, 893, 1359
Mhadhbi, N., 917
Michalak, T., 495, 500
Michel, L., 1354
Mikša, M., 259, 262, 273, 276, 279, 309
Miles, E., 688
Milgrom, P., 487
Mills, B. E., 19
Millstein, T., 79, 92, 769, 808
Minato, S., 32, 403
Minton, S., 42, 215
Mirchandani, P., 158
Mishchenko, A., 32, 403, 417, 561, 660, 750, 754, 1051
Misra, N., 709, 710, 722
Mistry, R., 1039
Mitchell, D. G., 40, 42, 133, 134, 213–215, 218, 222, 257, 277, 384, 438, 439, 456, 726, 860, 1336, 1352
Mitzenmacher, M., 38, 453
Miyano, E., 576
Miyazaki, S., 79, 93
Mladenov, M., 1039
Mneimneh, M. N., 585, 753, 1159, 1313
Moeller, R., 1224, 1237, 1242
Moffat, A., 652
Mohamedou, N. O., 912, 930, 939
Möhle, S., 143, 150
Mohnke, J., 561
Molitor, P., 561
Möller, A., 1274
Möller, D., 561
Möller, M. O., 797, 1274
Molloy, M. S. O., 41, 262, 277, 279, 443, 453, 456, 457, 608
Monasson, R., 40, 41, 151, 194, 223, 442, 451, 455, 867, 869–873, 876, 878, 879, 881, 882, 885, 887–890, 894
Mondadori, M., 407, 1047, 1048, 1224, 1237, 1240
Monien, B., 25, 356, 587, 600, 602, 603, 674, 681
Monma, C. L., 1279
Montanari, A., 224, 444, 446, 451, 454, 455, 869, 870, 872, 873, 877, 890, 892, 894, 895
Montmirail, V., 134, 158
Moon, C., 1069, 1074
Moondanos, J., 1063
Moore, C., 37, 41, 257, 277, 447, 467, 468
Moore, J. S., 1268
Mora, T., 224, 870, 872, 873
Morgado, A., 134, 155–160, 163, 323,

- 660, 914, 918, 919, 934, 935,
937, 938, 942, 954, 955, 961,
966, 967, 1124
- Morioka, T., 322, 1214
- Morozov, A., 498
- Morris, P., 213, 218
- Moses, Y., 1223–1226, 1228, 1247,
1248
- Moskał, M., 1285, 1311, 1312
- Moskewicz, M. H., 756
- Moskewicz, M. W., 24, 25, 116, 118–
121, 124, 126, 127, 133, 140,
142, 143, 147–149, 161, 162,
234, 241, 247, 254, 269, 385,
397, 398, 474, 477, 553, 562,
797, 1000, 1058, 1065, 1077,
1104, 1109, 1110, 1118, 1167,
1179, 1242, 1286, 1288, 1292,
1352, 1355
- Mossel, E., 224, 225, 451, 890, 892,
894
- Motter, D., 31, 107
- Motter, D. B., 418
- Motwani, R., 40, 464, 476, 879, 1336
- Moura, de, L., 287, 293, 756, 798,
836, 1273, 1285, 1286, 1291,
1308, 1309, 1311–1313
- Muhammad, R., 76
- Muise, C. J., 936
- Mukhopadhyay, A., 561
- Mull, N., 254, 723
- Müller, M., 251, 252, 267, 281, 319
- Mundhenk, M., 1334, 1335
- Muñoz, V., 918
- Murgai, R., 1069, 1074
- Murphy, K., 1009
- Murray, N. V., 1048
- Murta, C. D., 469
- Murty, K. G., 1029
- Musliu, N., 936
- Musuvathi, M., 751, 752, 1274, 1307,
1315
- Myllymäki, P., 937
- 635, 636, 660, 661, 968, 1026,
1274, 1302, 1313
- Nadel, B. A., 83
- Naik, M., 157, 938
- Nam, G.-J., 534
- Naor, A., 224, 441, 448
- Naor, J., 1281
- Narizzano, M., 1159, 1160, 1164, 1166–
1168, 1172, 1179, 1180, 1183,
1184, 1189, 1198, 1199, 1343,
1355, 1358, 1360
- Narodytska, N., 134, 155, 158, 159,
914, 937, 942, 957, 958, 1015,
1039, 1124
- Narváez, D., 636, 645, 655
- Natraj, N., 1117
- Nau, D., 765
- Nebel, B., 765, 770, 771, 1235, 1236
- Negrete, S., 918, 936
- Nelson, G., 803, 810, 1268, 1273, 1282,
1285, 1286, 1303, 1305, 1311,
1312
- Nemhauser, G. L., 965, 1115–1118
- Nepomnjaščii, V. A., 320
- Nesterov, Y., 311
- Neumann, A., 487
- Neves, M., 970, 971
- Newell, A., 19
- Newman, M. E. J., 722, 970, 1356,
1357
- Newman, N., 486, 487, 499
- Newsham, Z., 722
- Ng, A. Y., 1351
- Nguyen, L. A., 1228
- Nguyen, P., 322
- Nguyen, V.-H., 80, 81
- Nicolini, E., 1302, 1303
- Niedermeier, R., 43, 44, 694, 695,
697, 910
- Niemelä, I., 151, 396, 414, 740, 746,
770–772, 774, 775, 779, 1030,
1031, 1064
- Niemetz, A., 1285, 1315
- Niermann, T. M., 1067
- Nieuwenhuis, R., 81, 82, 134, 156,
157, 655, 851, 918, 936, 944,
1237, 1273, 1274, 1285, 1286,

- 1290, 1291, 1293–1296, 1300–
 1303, 1307
 Nightingale, P. W., 81, 134, 156
 Nilsson, N. J., 1338
 Nisan, N., 301, 306
 Nishimura, N., 471, 705, 706, 709,
 711, 994
 Niskanen, A., 937
 Nocco, S., 755
 Nohl, K., 150, 162, 414, 1016, 1030
 Nöhrer, A., 134, 159, 660
 Nonnengart, A., 1047
 Norden, van, L., 47
 Nordström, J., 134, 160, 235, 249,
 253, 255, 256, 258–260, 262–
 269, 271–277, 279–281, 287,
 293–300, 302, 304, 306–311,
 319, 726
 Nori, A. V., 938
 Novikov, Y., 116, 128, 148, 154, 162,
 244, 249, 385, 635, 642, 653,
 658, 1242
 Nowé, A., 498
 Nudelman, E., 474, 485, 490, 496,
 498, 968
 Nuzzo, P., 1315

 Obdrzálek, J., 721
 Ochremiak, J., 271, 272
 Oetsch, J., 1159
 Ogawa, T., 81, 134, 156, 944
 Oh, C., 148, 150, 162, 402
 Oh, Y., 585, 1313
 Ohrimenko, O., 160
 O’Kane, D., 869
 Okushi, F., 590, 599
 Olariu, S., 720
 Oliva, S., 319
 Oliveira, A. L., 938
 Oliveras, A., 81, 82, 134, 156, 157,
 655, 851, 906, 908, 910, 912,
 930, 939, 942, 944, 1237, 1273,
 1274, 1285, 1286, 1290, 1291,
 1293–1296, 1300, 1301, 1303,
 1307
 Olteanu, D., 1018
 Onn, S., 275, 281

 Oppacher, F., 1047
 Oppen, D. C., 1268, 1273, 1275, 1303,
 1305
 Ordyniak, S., 708–710, 720
 Orlin, J., 1114
 Orponen, P., 216, 224, 451, 881, 883
 Osama, M., 419
 Ostrowski, R., 75, 399, 417
 O’Sullivan, B., 492–494, 704, 708, 709
 Otpuschennikov, I. V., 151
 Otten, J., 76
 Ottosson, G., 1361
 Ou, X., 1285, 1286, 1292
 Ouaknine, J., 742, 745, 748, 798, 823,
 1274, 1285, 1310, 1311
 Oum, S.-i., 720
 Ouyang, M., 354, 355, 374, 377
 Owens, A. J., 42
 Owre, S., 1285

 Padberg, M., 965
 Pagnani, A., 895
 Pais, J., 1315
 Palassini, M., 872
 Palem, K., 40, 1336
 Palena, M., 750
 Paleo, B. W., 657, 658
 Palmer, R., 861
 Palti, A., 392, 1274, 1302
 Palyan, A., 906, 909
 Pan, G., 31, 107, 109, 269, 722, 1159,
 1171, 1172, 1191, 1223–1225,
 1251–1255, 1260
 Panchenko, D., 869
 Panda, B., 399
 Panda, S., 561
 Pandey, B., 784
 Pang, S., 254, 258
 Pankratov, D., 305, 306
 Papadimitriou, C. H., 26, 42, 43, 214–
 217, 451, 464, 571, 572, 580,
 672, 675–679, 682, 683, 686,
 725, 859, 878, 879, 943, 994,
 1049, 1157, 1158, 1273, 1279,
 1331, 1333, 1335
 Papavasileiou, V., 1315
 Pappas, G. J., 1315

- Pardo, A., 32
Paredes, R., 1015, 1039
Paris, J. B., 321, 322
Parisi, G., 214, 224, 454, 859, 866,
868, 869, 873, 876, 890, 893
Park, J. D., 936, 993
Park, T. J., 1349
Parkes, A. J., 93, 213, 1362
Parlak, A., 721
Parrain, A., 160, 286, 287, 309, 946,
947
Parrilo, P. A., 47, 311
Paruthi, V., 151, 416, 739, 1051, 1056,
1058, 1063
Parviainen, P., 930
Pasechnik, D. V., 45, 271, 311
Pasini, P., 750
Passerini, A., 1039
Patel, J. H., 1067
Patel-Schneider, P. F., 1223, 1224,
1237, 1239, 1240, 1242, 1243,
1247, 1291
Patti, D., 750
Patton, G. C., 835, 836
Paturi, R., 26, 233, 675, 679–681,
683–686, 697
Paul, W. J., 263–265
Paull, M., 33, 36, 222, 224, 437
Paulusma, D., 710, 720
Pawlowski, P. M., 797
Paxian, T., 945, 947
Pearl, J., 224, 1008, 1359, 1361
Pehoushek, J. D., 159, 996, 998, 1344
Peitl, T., 1183, 1206
Peled, D., 739, 740
Peltonen, J., 937
Pereira, F., 159, 937
Peres, Y., 37, 224, 440, 441, 448, 450,
864, 866
Pérez Cáceres, L., 488
Perron, L., 1354
Peterson, G., 1145
Petke, J., 80
Petrank, E., 1016, 1029
Petrie, K. E., 564
Peugniez, T. J., 80, 81, 92, 93, 156
Pfandler, A., 726
Pham, D.-N., 486, 769
Pham, D. N., 43, 76, 90, 213, 218,
219, 1048, 1064
Pham, T. A., 1018
Philipp, T., 134, 157, 415, 944, 945
Philips, A. B., 42, 215
Picard, J. C., 49
Pich, J., 1210, 1214
Pichora, M. C., 1274
Pierce, B. C., 591
Piette, C., 134, 158, 393, 408, 585
Pigorsch, F., 756
Pilarski, S., 208
Pilipczuk, M., 694, 695, 719
Pipatsrisawat, K., 116, 134, 147, 148,
160–162, 247, 252–255, 267,
392, 401, 406, 477, 906, 909,
998
Pippenger, N., 233, 263
Pistore, M., 1172
Pitassi, T., 38, 45, 222, 234, 235, 240,
251, 259, 267, 270, 272, 275,
276, 281, 300, 301, 305–307,
311, 318–321, 400, 443, 456,
717, 724, 889, 993, 994, 996,
999, 1000, 1331–1336, 1338,
1339, 1341, 1342, 1344–1346,
1349–1351, 1353, 1356, 1357
Pittel, B., 870
Pixley, C., 109, 782
Plaisted, D. A., 86, 417, 753, 756,
1047, 1160, 1171, 1172, 1193
Planes, J., 134, 157, 158, 660, 906,
908, 910, 912, 913, 917–919,
930, 939, 950, 952, 953, 968,
1123, 1124
Plass, M. F., 27, 51, 161, 403, 404,
672, 1153
Platania, L., 803
Plateau, G., 30
Plessier, B. F., 109, 561
Plis, S. M., 937
Plotkin, J. M., 36
Plummer, M. D., 725
Pnueli, A., 740, 742, 745, 765, 809,
1268, 1280, 1281
Podelski, A., 808, 811, 814

- Pólya, G., 360, 560
Pote, Y., 1038
Poupart, P., 150, 162, 269, 402
Pradhan, D. K., 144, 393, 401, 402, 417, 1066, 1069
Prasad, M., 740, 755, 756
Pratt, V., 1282
Prawitz, D., 19
Preiner, M., 1285, 1315
Prestwich, S. D., 80–82, 90, 91, 93, 134, 155, 156, 213, 218, 838, 839
Previti, A., 134, 155, 158–160, 963, 968
Profitlich, H. J., 1235, 1236
Proskurowski, A., 714
Prosser, P., 88, 118, 161, 299
Pudlák, P., 26, 260, 270, 272, 275, 276, 280, 304–306, 319, 320, 322, 680, 750–752, 1210, 1214, 1314
Puget, J.-F., 564
Puggelli, A., 1315
Pugh, W., 1284
Pulina, L., 1172
Purdom Jr., P. W., 24, 33, 34, 36, 213, 222, 380, 559, 562
Putinar, M., 48
Putnam, H., 19, 20, 75, 104, 111, 136, 138, 139, 141, 238, 240, 241, 392, 394, 396, 401, 644, 671, 674, 1102, 1103, 1170, 1236–1238, 1240, 1242
Qadeer, S., 770
Qardaji, W. H., 937
Qian, Z., 1015
Qiao, K., 937
Quan, Z., 158, 908, 918, 937
Queille, J.-P., 739
Quer, S., 750, 755
Quilbeuf, J., 1015
Quimper, C.-G., 92, 155
Quine, W. V. O., 19, 158, 514
Rabe, M. N., 134, 158, 1039, 1195, 1204
Rabinovitz, I., 802
Rabung, J. R., 834
Raddaoui, B., 156, 917
Rado, R., 833, 834
Radziszowski, S., 832
Ragde, P., 471, 705, 706, 709, 711, 994
Raghavan, P., 26, 43, 214, 464, 476, 676–679, 686
Raghothaman, M., 1315
Rahardja, S., 561
Rahwan, T., 495, 500
Raimi, R., 749, 799
Rajamani, S. K., 770, 805, 808, 809, 811, 812
Rajan, S. P., 802
Ramakrishnan, K., 214
Raman, V., 43, 44, 703, 910
Ramani, A., 43, 50, 532, 534, 563, 564, 849, 947
Ramanujan, M. S., 708, 709, 722
Ramesh, H., 45
Ramírez, M., 906, 909
Ramsey, F. P., 832
Ranise, S., 1274, 1286, 1292, 1302, 1303, 1306–1309
Ranjan, R., 109
Rankooh, M. F., 781
Rao, A., 301, 306
Rao, M. R., 28, 49
Rao, S., 968
Rapicault, P., 311, 936
Rasmussen, C., 1204
Rasmussen, R. A., 1066
Ratliff, H. D., 49
Ravaghan, P., 879
Ravve, E. V., 710, 716, 720, 721, 994
Raz, R., 234, 256, 260, 263
Razborov, A. A., 251, 254–256, 258, 260, 262, 265–267, 272, 274–276, 279, 311, 319–321, 442, 726
Razgon, I., 704, 708, 709
Rebola-Pardo, A., 160, 315, 415, 643
Reckhow, R. A., 121, 234, 313, 316, 639
Reed, B. A., 24, 36, 40, 224, 438, 441, 719

- Régin, J.-C., 1354
Reif, J., 1145
Reimer, S., 945, 947, 1198, 1199
Reiter, M. K., 1015
Reiter, R., 160, 963
Remshagen, A., 1352, 1358
Renegar, J., 369
Rényi, A., 1033
Resende, M. G. C., 213, 214
Reynolds, A., 1315
Rhodes, M., 271
Ricardo, Z., 454
Ricca, F., 957, 959
Ricci-Tersenghi, F., 224, 444, 446,
447, 454, 455, 869–873, 877,
878, 892, 894, 895
Rice, J. R., 489
Richardson, T., 877
Riis, S., 257, 260, 275, 320
Rijke, de, M., 1224
Ringeissen, C., 1272, 1303, 1306
Rintanen, J., 157, 159, 768–772, 774,
775, 779, 781, 782, 784, 1158,
1168
Rish, I., 105, 107, 213, 466
Ritirc, D., 282
Rivest, R. L., 670
Rivoire, O., 872
Robere, R., 272, 281, 303, 305–307
Robertson, N., 579, 613, 619, 622,
718, 721
Robinson, A., 1163
Robinson, J. A., 20, 21, 102, 138,
238, 395, 405, 638, 642, 1179
Robinson, N., 769
Robson, J. M., 233
Robson, R., 356
Rock, G., 1047
Rodeh, Y., 1268, 1280
Rodríguez-Carbonell, E., 81, 82, 134,
156, 157, 655, 944
Roig, C., 918, 936
Rolf, D., 679, 686
Rollini, S. F., 657
Romashchenko, A. E., 312
Romero, F. P., 938
Ron-Zewi, N., 279
Rong, Z., 938
Rose, D. J., 719
Rosenberg, I. G., 50
Rosenblitt, D., 769
Rosenbluth, A. W., 1004
Rosenbluth, M. N., 1004
Rosenthal, E., 1048
Rosenthal, J. W., 36
Ross, S., 1032, 1034
Rossi, C., 214
Rossman, B., 272, 320
Rossmanith, P., 43, 44, 910
Rossum, van, P., 1285, 1286, 1290–
1292, 1302, 1303, 1307–1309
Roth, D., 712, 993, 1015, 1334, 1335,
1351
Roth, J. P., 1067
Rothschild, B., 832
Rotics, U., 715, 720
Roussel, O., 134, 151, 155–157, 487,
497, 944, 945, 1118–1121
Roveri, M., 1172, 1223
Rowley, A. G. D., 1159, 1168, 1352,
1355, 1358
Roy, A., 92, 151, 562, 563, 565, 849
Roy, J. A., 417
Royle, G., 823
Rozenberg, G., 720
Ruan, Y., 474
Rubinstein, R. Y., 1005
Rubio, A., 1302
Rucinski, A., 863, 874
Rudeanu, S., 49, 50, 1087
Rudell, R. L., 31
Rueß, H., 756, 797, 1285, 1286, 1291,
1297, 1299
Rümmele, S., 726
Ruszczek, A. P., 1343, 1347
Rusinowitch, M., 1302
Russell, A., 822
Russell, B., 13
Rutenbar, R. A., 43, 534, 917, 937
Ryan, L., 116, 120, 125, 160, 247,
254
Rybalchenko, A., 814, 1315
Ryser, H. J., 611, 620
Ryvchin, V., 134, 143, 150, 153, 158,

- 245, 392, 402, 635, 636, 660,
661
- Sabharwal, A., 23, 119, 121, 122, 127,
134, 141, 142, 147, 159, 160,
224, 225, 241, 251, 257, 258,
371, 392, 471, 492–494, 497,
559, 562, 564, 646, 712, 1006–
1009, 1016, 1020, 1023, 1025,
1026, 1036, 1037, 1172
- Sacerdoti, E. D., 769
- Sæther, S. H., 720
- Saetti, A., 483
- Safarpour, S., 748, 755, 917, 937, 1061,
1062
- Säflund, M., 109
- Sagiv, M., 134, 158
- Saïdi, H., 805
- Saidi, M. R., 558
- Saigal, R., 45
- Saikko, P., 134, 158, 937, 940, 961,
963–967
- Sais, L., 75, 134, 143, 155, 156, 159,
213, 219, 393, 399, 408, 417,
564, 849, 917, 1089, 1361
- Sakai, M., 287
- Sakallah, K. A., 24, 25, 43, 50, 109,
116, 118–121, 124, 126, 133,
134, 137, 140, 142–144, 148,
149, 151, 160–162, 234, 241,
268, 287, 385, 392, 530–532,
534, 546, 548, 549, 552, 553,
562–565, 585, 601, 644, 660,
750, 753, 755, 849, 917, 937,
947, 1026, 1064, 1065, 1106,
1108–1110, 1112, 1118, 1159,
1167, 1179, 1242, 1274, 1285,
1302, 1313
- Saks, M. E., 29, 38, 222, 443, 456,
675, 679, 686, 889, 1345
- Saldanha, A., 133, 1069, 1074
- Salhi, Y., 134, 155, 159
- Samer, M., 705, 706, 710, 714–717,
719, 721, 1182, 1183
- Samorodnitsky, G., 466
- Samson, E. W., 19
- Samuels, B., 498
- Samulowitz, H., 142, 492–494, 497,
1168, 1198, 1358, 1360
- Sang, T., 121, 159, 400, 993, 996,
1000, 1344, 1349, 1357
- Sangiovanni-Vincentelli, A. L., 108,
133, 417, 1064, 1069, 1071,
1072, 1074, 1102, 1115, 1315
- Sanner, S., 1039
- Santuari, A., 1309
- Sar, C., 803
- Sarfert, T., 1069
- Sarkhel, S., 1018
- Sassano, A., 585
- Sattar, A., 43, 76, 90, 213, 218, 769,
1048, 1064
- Sattler, U., 1223–1225, 1251–1255,
1260
- Saubion, F., 214
- Saurabh, S., 694, 695, 708, 709, 722
- Savage, J. E., 263, 265
- Savický, P., 617
- Savitch, W. J., 51, 740, 755, 1137
- Savoj, H., 108, 1069, 1074
- Saxe, J. B., 803, 810, 1211, 1282,
1285, 1286, 1292, 1311, 1312
- Saxena, P., 1015, 1039
- Say, B., 311
- Scarcello, F., 714, 721, 722, 994
- Schädlich, F., 443
- Schaefer, T. J., 27, 51, 671, 673, 706,
1138, 1139
- Schaerf, A., 157
- Schaerf, M., 755, 1159, 1162, 1168,
1172, 1184–1186, 1199, 1200,
1338, 1342
- Schapiro, R. E., 190, 416
- Schaub, T., 151, 160, 162, 287, 487,
493, 494, 497, 498, 954, 1159
- Scheder, D., 617
- Schevon, C., 215
- Schiermeyer, I., 26, 373
- Schiex, T., 43, 911, 930, 1107
- Schild, K. D., 1223, 1229–1231
- Schilling, C., 938
- Schlipf, J. S., 28, 31, 107, 383, 395,
418, 626
- Schlöffel, J., 1072, 1077

- Schlotter, I., 701
 Schmee, J., 491
 Schmidt, R. A., 1223, 1224
 Schmidt-Schauß, M., 1229
 Schmitt, P., 54, 1146
 Schneider-Kamp, P., 154, 635
 Schneider, M., 483
 Schneider, P. R., 561
 Schnoor, H., 673
 Schnorr, C.-P., 233
 Scholl, C., 153, 392, 413, 756, 1159
 Schöning, U., 26, 43, 92, 214, 487,
 676–679, 686, 879
 Schonlau, M., 486
 Schrag, R. C., 116, 118, 119, 121,
 161, 234, 998, 999, 1242, 1342–
 1344, 1358
 Schrijver, A., 44, 45, 311
 Schubert, L., 769
 Schuler, R., 27, 43, 680, 682, 879
 Schulz, M. H., 161, 1069
 Schulz, S., 1285, 1290–1292, 1302
 Schuppan, V., 740, 741, 745–747, 753
 Schuurmans, D., 42, 213, 218, 219
 Schwefel, H.-P., 42
 Schwendimann, S., 1223, 1235, 1236
 Scutella, M. G., 28
 Sebastiani, R., 76, 133, 160, 756, 779,
 851, 1048, 1062, 1064, 1172,
 1223, 1224, 1234, 1235, 1237,
 1239, 1240, 1242–1244, 1248,
 1256–1258, 1268, 1274, 1285,
 1286, 1289–1293, 1302, 1303,
 1307–1309, 1313–1315
 Seese, D., 719
 Segal, I., 487
 Segerlind, N., 235, 312, 319, 999, 1209,
 1344
 Seguin, C., 937
 Seidl, M., 160, 315, 393, 410, 413,
 640–642, 644, 645, 660, 1135,
 1159, 1160, 1172, 1177, 1183,
 1184, 1190, 1191, 1198–1201,
 1203, 1205, 1213, 1214
 Seiferas, J. I., 688
 Seipp, J., 497
 Seitz, S., 216, 224, 451, 881, 883
 Selensky, E., 838, 839
 Sellmann, M., 142, 485, 491–494, 496,
 497, 844, 968, 1007
 Selman, B., 40–42, 76, 90, 91, 93,
 126, 127, 133, 134, 140, 147,
 148, 151, 157, 159–161, 194,
 213–216, 218, 222–225, 401,
 438, 439, 442, 456, 464–468,
 470–476, 490, 704, 707, 755,
 765, 766, 769, 770, 775, 777,
 860, 878, 881, 885, 1003, 1004,
 1006–1009, 1016, 1020, 1023,
 1025, 1026, 1036, 1037, 1048,
 1159, 1172, 1242, 1336, 1347,
 1352
 Semenov, A. A., 151
 Semerjian, G., 224, 444, 446, 451,
 454, 455, 869, 870, 872, 873,
 877, 879, 881, 882, 892, 894,
 895
 Sentovich, E., 1069, 1074
 Servedio, R., 320
 Seshia, S. A., 133, 134, 158, 160, 723,
 798, 851, 1036–1039, 1159,
 1204, 1256, 1274–1276, 1282–
 1285, 1290, 1310, 1311, 1315
 Sethi, D., 158
 Sethi, R., 1273
 Seyfried, M., 1235, 1236
 Seymour, P. D., 579, 612, 613, 619,
 622, 718, 721
 Sgall, J., 260, 272, 275, 276, 280, 320,
 617
 Shabani, A., 937
 Shacham, O., 657
 Shachter, R. D., 1335
 Shader, B. L., 614, 617, 618
 Shah, D., 890
 Shamir, A., 27, 587, 601
 Shang, Y., 42, 213, 219, 221
 Shankar, N., 1285, 1297, 1299
 Shannon, C. E., 18, 560
 Shaohan, M., 575
 Shapire, R. E., 90
 Sharir, M., 809
 Sharygina, N., 657, 755, 797, 808–
 811, 1159

- Shaw, A., 401
 Shaw, J., 19
 Sheeran, M., 24, 134, 203, 417, 749, 1048
 Sheini, H. M., 50, 287, 1106, 1112, 1274, 1285, 1302
 Shen, H., 44, 906, 908, 910
 Shen, S., 1015, 1039, 1313
 Sherali, H. D., 44, 311
 Sheridan, D., 745, 1047, 1160
 Sherry, J., 936
 Shi, J., 1072
 Shikanian, I., 1275
 Shimono, T., 161, 1065, 1068, 1076, 1077
 Shin, J.-A., 779, 781
 Shinde, S., 1039
 Shine, S., 1015, 1039
 Shirai, Y., 820
 Shliffer, E., 49
 Shlyakhter, I., 803, 814
 Shmoys, D., 822, 823
 Shoham, Y., 474, 485, 490, 496, 498, 968
 Shokrollahi, M. A., 453
 Shor, N. Z., 311
 Shor, P., 1349
 Shoukry, Y., 1315
 Shrikhande, S. S., 824
 Shrotri, A. A., 1015, 1017, 1019, 1032, 1034, 1035, 1039
 Shtrichman, O., 1268
 Shukla, A., 1210, 1213
 Si, X., 157, 938
 Sideri, M., 714, 722, 994
 Siegel, M., 1268, 1280
 Siegel, P., 396
 Sieveking, M., 1279
 Sievers, S., 497
 Sifakis, J., 739
 Sikdar, S., 703
 Silva, F., 937
 Silveira, L. M., 151
 Silverthorn, B., 483
 Simeone, B., 50, 1093
 Simon, H., 19
 Simon, L., 107, 127, 134, 148, 153, 160–162, 247, 268, 269, 313, 392, 403, 409, 487, 636, 722, 918, 919, 935
 Simons, P., 396, 746, 1058
 Sinclair, A., 148, 224, 473, 1015
 Singer, J., 93
 Singh, D. P., 1159
 Singh, K., 1069, 1074
 Singh, R., 1315
 Singh, S., 749
 Singla, P., 1018
 Sinha, N., 1281
 Sinz, C., 81, 107, 134, 148, 151, 162, 312, 313, 392, 400, 401, 415, 417–419, 644, 657, 660, 944, 1056, 1090
 Sipser, M., 1211
 Sistla, A. P., 740, 745, 1137
 Slaney, J. K., 43, 151, 213, 218, 471, 472, 820, 822, 826, 827, 829, 831, 849, 1006
 Slepian, D., 560
 Slivovsky, F., 417, 720, 724, 1182, 1183, 1206
 Sloan, R. H., 578, 617
 Smaill, A., 93
 Smith, B. M., 86, 88, 465, 564
 Smith, D., 748, 755
 Smith, G. L., 1066
 Smith, J. E., 214
 Smolensky, R., 321
 Smolka, G., 1229
 Smullyan, R. M., 1235
 Smyth, K., 1168, 1352, 1358
 Snoek, J., 483
 Sofronie-Stokkermans, V., 1315
 Soh, T., 158
 Soininen, T., 396, 746
 Sokolov, D., 266, 272, 274, 280, 281, 303, 305, 312
 Solar-Lezama, A., 1315
 Soler, J. R., 915, 918, 919, 936
 Somenzi, F., 31, 32, 107, 254, 393, 399, 402, 406, 408, 409, 561, 655, 750, 755
 Soos, M., 150, 162, 401, 414, 416, 418, 636, 1016, 1020, 1023,

- 1030–1032, 1039
 Sorea, M., 756, 1285, 1286, 1291
 Sörensson, N., 50, 82, 116, 126, 134, 144, 145, 147, 150, 153, 155–157, 161, 162, 287, 385, 392, 399, 403, 407, 417, 489, 635, 636, 645, 646, 653, 654, 661, 748, 750, 845–847, 849, 850, 942, 944, 947, 1009, 1051, 1118–1120, 1242
 Sörényi, B., 578, 617
 Sorkin, G. B., 24, 37, 44, 224, 440, 441, 453
 Southey, F., 42, 213, 218, 219
 Spears, W. M., 213
 Speckenmeyer, E., 25, 189, 207, 356, 392, 587, 600, 602, 603, 674, 681
 Spence, I., 259
 Spencer, J. H., 832, 864, 870
 Spinrad, J. P., 701
 Spirakis, P., 40, 1336
 Springenberg, J. T., 483
 Sreenivasaiah, K., 1210
 Srivastava, P., 1015
 St-Aubin, R., 1348
 Stachniak, Z., 76, 1064
 Stallman, R. M., 118, 119, 161, 1107
 Stallmann, M. F. M., 213
 Stålmarmark, G., 24, 109, 203, 261, 417, 749, 1048
 Stamatiou, Y. C., 40, 224, 440, 1336
 Stanion, T., 371
 Stata, R., 803
 Steiglitz, K., 215, 859
 Stein, C., 670
 Steinke, P., 134, 157, 487, 944, 945
 Stephan, P., 1064, 1069, 1071, 1072, 1074
 Stephan, P. R., 133
 Stern, R., 160
 Steurer, D., 311
 Stevens, B., 836
 Stickel, M. E., 31, 149, 161, 397, 398, 826, 827, 829–831, 849, 1356
 Stinson, D., 836
 Stockmeyer, L. J., 51, 781, 1008, 1016, 1018, 1020, 1024, 1026, 1137, 1138
 Stoffel, D., 797
 Stone, M. H., 16
 Storey, T. M., 1066
 Streichert, T., 1092
 Strichman, O., 134, 153, 158, 392, 393, 396, 404, 406, 636, 657, 660, 742, 745, 748, 750, 798, 802, 1062, 1256, 1274, 1280, 1281, 1283–1285, 1290, 1310, 1311, 1313
 Stuckey, P. J., 76, 80, 151, 160, 311, 968, 1281
 Stump, A., 1274, 1285, 1286, 1290, 1291, 1297, 1301, 1303, 1307, 1308
 Stutz, J., 213, 216
 Stützle, T., 43, 213, 218, 488, 1352
 Su, K., 906, 908, 968
 Subbarayan, S., 144, 393, 401, 402
 Subramani, K., 54, 1143, 1282
 Subramanian, D., 456
 Suci, D., 718
 Suda, M., 151, 162, 393
 Sudan, M., 51, 673, 1139
 Suen, E., 1047
 Suen, S., 37, 40, 224, 440, 447, 453, 454, 456, 884, 888
 Sun, X., 29, 50, 708
 Sundaram, R., 822
 Sussman, G. J., 118, 119, 161, 1107
 Suy, J., 918, 936
 Swaminathan, R. P., 28, 395
 Sward, G. L., 396
 Swernofsky, J., 266, 280
 Szeider, S., 30, 322, 471, 572, 573, 576, 577, 579, 596, 597, 616, 617, 697, 698, 701, 703–712, 714–726, 994, 1182, 1183, 1206
 Szemerédi, E., 22, 23, 38, 222, 257, 438, 442, 443, 456
 Szepesvári, C., 489
 Tabuada, P., 1315
 Tacchella, A., 1062, 1159, 1160, 1164, 1166–1168, 1172, 1179, 1180,

- 1183, 1184, 1189, 1224, 1239,
 1242–1244, 1247, 1248, 1292,
 1343, 1355, 1358, 1360
 Tafertshofer, P., 1064, 1071
 Taga, A., 80
 Tak, van der, P., 970, 1268, 1297
 Talagrand, M., 869
 Talupur, M., 1279, 1281
 Tamaki, S., 27, 679
 Tamir, A., 1281
 Tamura, N., 80, 158
 Tan, L.-Y., 320
 Tang, B., 265, 266, 268, 280, 304
 Tangmunarunkit, H., 530
 Tanjo, T., 80
 Taqqu, M., 466
 Tarim, S. A., 1353–1355, 1362
 Tarjan, R. E., 27, 51, 109, 161, 264,
 265, 403, 404, 412, 672, 719,
 1153, 1273
 Tarski, A., 16
 Tarui, J., 688
 Tatikonda, S., 890
 Taylor, W. M., 40, 222
 Tchinda, R. K., 322
 Teich, J., 1092
 Telle, J. A., 720
 Teller, A. H., 1004
 Teller, E., 1004
 Tentrup, L., 134, 158, 1195, 1204
 Terra-Neves, M., 970
 Tessaris, S., 1235, 1236
 Thapen, N., 251, 262, 266, 279, 280,
 304, 306, 315, 322, 641, 643
 Theodoropoulos, P., 1020, 1023, 1025,
 1026, 1028
 Théry, L., 635
 Thiébaux, S., 151, 471, 1006
 Thierauf, T., 687
 Thiffault, C., 76, 77, 151, 1059, 1062
 Thilikos, D. M., 719
 Thirumalai, D., 869
 Thomas, C., 150, 162
 Thomas, R., 579, 613, 619, 622
 Thomas, W., 1269, 1272
 Thornton, C., 483
 Thornton, J. R., 43, 76, 90, 213, 218,
 219, 968, 1048, 1064
 Thorsteinsson, E. S., 1361
 Thurley, M., 134, 160, 161, 252, 253,
 255, 267, 996, 1000
 Tierney, K., 485, 491, 492, 494, 496,
 968
 Tilak Raj, S., 936
 Tille, D., 1072, 1077
 Tinelli, C. M., 133, 160, 851, 1237,
 1272, 1274, 1275, 1285, 1286,
 1290, 1291, 1293–1296, 1300,
 1301, 1303–1306, 1311, 1312,
 1315
 Tischner, D., 938
 Tiwari, A., 1273, 1315
 Tiwari, S., 305, 306
 Toda, S., 995, 1017
 Todinca, I., 719
 Toft, B., 613
 Tomov, N., 93
 Tompits, H., 393, 410, 641, 642, 1135,
 1159, 1160
 Tompkins, D. A. D., 42, 213, 218,
 219, 492
 Topcu, U., 936
 Torán, J., 262
 Torlak, E., 157, 1315
 Torres-Jiménez, J., 937
 Touati, H. J., 32, 108
 Tovey, C. A., 29, 617
 Trager, B. M., 561
 Traverso, P., 31
 Trentin, P., 1315
 Tretkoff, C., 467
 Treybig, L. B., 1279, 1282, 1283
 Trick, M., 476
 Trimble, J., 299
 Trindade, A. B., 937
 Trischler, E., 1069
 Troyansky, L., 40, 41, 151, 194, 223,
 442, 878, 885
 Truemper, K., 29, 616
 Truszczyński, M., 590, 594, 598, 599,
 603, 604, 607, 616, 622, 833–
 835, 850
 Tsai, C.-C., 561
 Tseitin, G. S., 20, 22, 76, 239, 240,

- 312, 409, 415–417, 636, 639,
 642, 743, 746, 1007, 1135,
 1160, 1193
 Tsimpoukelli, M., 134, 158
 Tsiskaridze, N., 1315
 Tsitovich, A., 657
 Tsitsiklis, J. N., 1335
 Tsuji, Y. K., 377, 378, 404
 Tu, K., 1191
 Tulsiani, M., 311
 Tuncel, L., 44
 Turán, G., 282, 285, 293, 302, 321,
 578, 617, 644
 Turing, A. M., 15
 Turner, H., 782, 1159
 Turquette, A., 16
 Turquette, F., 16
 Turquette, M., 16
 Tveretina, O., 312
 Twohey, P., 803
 Tzameret, I., 311, 318

 Udupa, A., 1315
 Ullman, J. D., 527, 552
 Umans, C., 1346
 Upfal, E., 38, 224, 453, 867
 Urban, A., 938
 Urbanke, R., 877
 Uribe, T. E., 31, 805
 Urquhart, A., 22, 23, 240, 256, 257,
 322, 415, 534, 644

 Valiant, L. G., 263, 685, 995, 1004,
 1008, 1016, 1018, 1019, 1039
 Vallati, M., 483
 Valmari, A., 770
 Vamvakari, M., 224
 Van den Broeck, G., 1039
 Van Gelder, A., 29, 30, 41, 76, 79,
 92, 127, 154, 244, 249, 251,
 254, 259, 377, 378, 404, 405,
 408, 589, 599, 608, 635, 636,
 638, 646, 649, 654, 657, 724,
 754, 1177, 1181–1183, 1189,
 1198, 1201, 1205, 1206, 1210,
 1213, 1349
 Van Hentenryck, P., 1354

 Vanfleet, W. M., 31, 107, 383, 418,
 626
 Vardi, M. Y., 31, 107, 109, 158, 159,
 269, 311, 312, 456, 683, 713,
 740, 745, 1015–1020, 1023,
 1025–1028, 1030, 1032, 1034–
 1039, 1159, 1171, 1172, 1191,
 1223–1225, 1228, 1229, 1248–
 1255, 1260
 Vasquez, M., 213
 Vatshelle, M., 720
 Vazirani, V. V., 685, 1004, 1008, 1016,
 1018, 1019, 1032, 1034, 1039
 Vecchi, M. P., 42, 218, 1004
 Veith, H., 719, 756, 805, 808, 1194
 Velev, M. N., 77, 80, 534, 1047, 1256,
 1268, 1278, 1283
 Vendramineto, D., 750
 Veneris, A. G., 158, 748, 755, 917,
 937, 1061, 1062, 1159

 Venn, J., 11
 Venugopal, D., 1018
 Verfaillie, G., 1107
 Vescovi, M., 1223, 1224, 1256–1258
 Vidunas, L. J., 1066
 Vigneron, L., 1273
 Vila, L., 466
 Vilenchik, D., 451, 894
 Villa, T., 1102, 1115
 Villafiorita, A., 1224, 1234, 1237, 1242
 Villanger, Y., 719
 Villaret, M., 918, 936
 Vinay, V., 374, 377, 1342
 Vinyals, M., 134, 160, 253–256, 262,
 266, 268, 272, 277, 279, 293,
 296–299, 302, 306–308, 310,
 319
 Viola, E., 688
 Virasoro, M. A., 859, 866, 868, 869,
 873
 Vizel, Y., 134, 635, 636, 661
 Vollmer, H., 673
 Vorobjov, N., 311
 Voronkov, A., 134, 160, 1224
 Vossen, T., 765
 Vroon, D., 417
 Vušković, K., 28

- Živný, S., 267, 268
- W. Li, F. P., D. Wu, 843
- Wagner, D. K., 28
- Wah, B. W., 42, 213, 219, 221, 380
- Wahlström, M., 373, 682, 698, 706
- Wainwright, M. J., 224, 225, 890, 892
- Wakrime, A. A., 938
- Wallace, R. J., 44, 498, 907
- Wallner, J. P., 134, 158, 937
- Wallon, R., 298
- Walser, J. P., 213, 765, 1362
- Walsh, M. J., 42
- Walsh, T., 42, 76, 77, 92, 93, 134, 151, 155, 157, 213, 219, 465, 467, 471, 472, 474, 476, 745, 820, 1006, 1059, 1062, 1342, 1343, 1353–1355, 1357, 1362
- Wang, C., 814, 1292
- Wang, J., 24, 377, 380, 703, 1342
- Wang, L.-C., 1063
- Wang, Y., 44
- Wang, Y.-S., 1039
- Wang, Z., 311
- Ward, J., 31, 107
- Warners, J. P., 46, 81, 82, 111, 151, 155, 157, 203, 375, 416, 417, 609, 944, 945, 1090, 1118
- Watanabe, O., 43
- Weaver, S. A., 31, 107, 383, 418, 626
- Weber, M., 561
- Weber, T., 154, 635, 823
- Wedler, M., 281, 797
- Wegener, I., 214
- Wegman, M. N., 801, 836, 1016, 1025
- Wei, W., 90, 91, 93, 213, 1003, 1004, 1016
- Wei, Y., 251, 267
- Weidenbach, C., 1047, 1224
- Weigt, M. E., 442, 869, 870, 872, 873, 878, 879, 881, 882, 885, 895
- Weiss, Y., 876, 890, 1009
- Weissenbacher, G., 134, 158, 792, 814, 938, 1315
- Weisz, G., 489
- Welch, W. J., 486
- Weld, D. S., 79, 92, 765, 769, 779, 1247, 1248, 1285, 1286, 1290–1292, 1362
- Welp, T., 417
- Werner, B., 635
- Wetzler, N., 154, 249, 251, 313, 315, 322, 411, 635, 642, 645, 650, 652, 657, 1179
- Whittemore, J. P., 392, 750
- Wickramaarachchi, G. T., 937
- Widebäck, F., 109
- Widl, M., 1181, 1182, 1189, 1205, 1206, 1210, 1211, 1213
- Wienand, O., 281
- Wieringa, S., 409, 419, 585, 645
- Wigderson, A., 23, 38, 239, 257, 259–263, 274–276, 278, 279, 304, 1209, 1345
- Wijs, A., 419
- Wilkie, A. J., 321, 322
- Willems, B., 809
- Williams, B. C., 119
- Williams, C., 465, 466
- Williams, H. P., 1111
- Williams, M. J. Y., 1067
- Williams, P. F., 753, 756, 1051
- Williams, R., 151, 468, 470, 475, 687, 688, 704, 707
- Williams, T. W., 1067
- Williamson, D. P., 45, 46, 48, 917, 940, 965
- Wilson, D. B., 41, 224, 441, 877
- Wimmer, R., 1198, 1199
- Winter, F., 936
- Winter, J., 393, 404, 406, 644, 702, 1242
- Wintersteiger, C. M., 419
- Witten, I. H., 652
- Wittgenstein, L., 16
- Woeginger, G. J., 696
- Wolfe, D., 571, 572, 580, 725
- Wolfman, S. A., 765, 779, 1247, 1248, 1285, 1286, 1290–1292, 1362
- Wolfram, S., 476
- Wolkowicz, H., 45
- Wollan, P., 262, 279
- Wolper, P., 745, 809

- Wolpert, A., 27, 680
Wolsey, L. A., 965, 1115–1118
Woltran, S., 1135, 1159, 1160, 1172
Wood, D. E., 1113
Wood, S. B., 1198
Woods, A. R., 319, 321
Wormald, N. C., 36, 870
Wos, L., 820
Wotzlaw, A., 392
Wrathall, C., 51, 1138
Wrightson, G., 576, 712, 726
Wu, C.-A., 1059
Wu, D., 843, 844
Wu, W., 968
Wu, Z., 42, 213, 221, 722
- Xia, G., 696, 706
Xiao, F., 145, 150, 402, 408, 409
Xie, Y., 803
Xing, Z., 906, 909, 910, 913
Xu, C., 703
Xu, D., 576
Xu, H., 43, 917, 937
Xu, K., 908, 937
Xu, L., 159, 483, 485–487, 490, 491,
494–497, 499, 968
Xu, R., 908, 937
Xue, Y., 159, 1015
- Yamins, D., 483
Yan, J., 838
Yang, J., 803
Yannakakis, M., 109, 683, 714, 719,
917
Yao, A. C., 688
Yap, R. H. C., 1281
Yedidia, J. S., 876, 890, 1009
Yehudayoff, A., 296, 298, 301, 306,
308
Yeo, A., 703
Yin, J., 837, 843
Yokoo, M., 93
Yorav, K., 802, 808
Yorsh, G., 751, 752, 1315
Young, A., 560
Yu, H., 844
Yu, Y., 1285, 1290
Yun, D. Y. Y., 561
- Zabih, R., 136, 395, 1102, 1103
Zadeck, F. K., 801
Zaikin, O., 151
Zamponi, F., 894
Zane, F., 26, 675, 679, 680, 683, 684,
686, 697
Zantema, H., 31, 312
Zarba, C. G., 1273, 1303, 1305, 1306,
1314, 1315
Zazon-Ivry, M., 81, 134, 156
Zdeborová, L., 224, 444, 446, 454,
895
Zecchina, R., 40, 41, 151, 194, 214,
222–225, 383, 442, 444, 867,
869–873, 875, 877, 878, 885,
890–893, 895, 1359
Zeitouni, O., 868
Zemor, G., 836
Zeng, Z., 1362
Zhang, H., 44, 133, 149, 161, 213,
397, 398, 474, 755, 819, 821,
825–831, 837, 841–844, 848–
851, 906, 908, 910, 937, 1104,
1356
Zhang, J., 820, 830, 831, 838, 849,
850, 1313
Zhang, J. S., 561
Zhang, L., 24, 25, 116, 118–121, 124–
127, 133, 134, 140, 142, 143,
147–149, 154, 161, 162, 234,
241, 247, 254, 269, 385, 397,
398, 474, 477, 553, 562, 585,
635, 645, 653, 654, 754, 756,
797, 810, 812, 918, 936, 1000,
1058, 1065, 1077, 1104, 1109,
1110, 1118, 1166, 1167, 1172,
1179–1181, 1184, 1189, 1240,
1242, 1281, 1286, 1288, 1292,
1313, 1352, 1355
Zhang, T., 947
Zhang, W., 44, 722, 906, 909, 910,
913
Zhang, X., 157, 401, 938
Zhang, Y., 843, 844, 1015
Zhang, Z., 1061
Zhao, S., 1020, 1026
Zhao, X., 44, 54, 573, 574, 576–580,

- 582–584, 711, 1143, 1144,
1152
- Zhao, Y., 24, 25, 116, 126, 127, 133,
140, 142, 143, 147–149, 161,
162, 234, 241, 247, 254, 269,
385, 397, 398, 474, 477, 553,
562, 756, 797, 1000, 1058,
1065, 1077, 1104, 1118, 1179,
1242, 1292, 1352, 1355
- Zheng, Y., 1315
- Zhou, Z., 1015
- Zhu, C. S., 158, 938
- Zhu, L., 821, 825, 829, 840, 843
- Zhu, Q., 417
- Zhu, Y., 32, 133, 157, 159, 371, 739,
740, 742, 744, 745, 747–749,
753, 755, 756, 765, 799, 1171,
1172
- Zhu, Z., 158, 918, 919, 935
- Zhuang, Y., 1358
- Zivny, S., 709, 710
- Zolda, M., 1135, 1159, 1160
- Zucchelli, D., 1302, 1303
- Zuckerman, D., 148, 473
- Zumstein, P., 617
- Zwick, U., 46, 917
- Zwieten, van, J. E., 191, 197, 201,
203, 1030

Contributing Authors and Affiliations

Dimitris Achlioptas <i>University of Athens Athens, Greece</i>	437	Rolf Drechsler <i>University of Bremen / DFKI Bremen, Germany</i>	1047
Fabrizio Altarelli	859	John Franco <i>University of Cincinnati Cincinnati, OH, USA</i>	3
Fahiem Bacchus <i>University of Toronto Toronto, Canada</i>	929	Enrico Giunchiglia <i>University of Genoa Genoa, Italy</i>	1157
Clark Barrett <i>Stanford University Stanford, CA, USA</i>	1267	Carla P. Gomes <i>Cornell University Ithaca, NY, USA</i>	463, 993
Olaf Beyersdorff <i>University of Jena Jena, Germany</i>	1177	Marijn Heule <i>Carnegie Mellon University Pittsburgh, PA, USA</i>	183, 635
Armin Biere <i>Johannes Kepler University Linz, Austria</i>	391, 739	Edward A. Hirsch <i>Steklov Institute of Mathematics at St. Petersburg / St. Petersburg State University St. Petersburg, Russia</i>	669
Uwe Bubeck <i>Sunnyvale, CA, USA</i>	1131	Holger H. Hoos <i>Leiden University / University of British Columbia The Netherlands / Canada</i>	481
Sam Buss <i>University of California San Diego La Jolla, CA, USA</i>	233	Frank Hutter <i>University of Freiburg / Bosch Freiburg, Germany</i>	481
Supratik Chakraborty <i>Indian Institute of Technology Bombay Mumbai, India</i>	1015	Mikoláš Janota <i>Czech Technical University Prague, Czech Republic</i>	1177
Evgeny Dantsin <i>Roosevelt University Chicago, IL, USA</i>	669	Matti Järvisalo <i>University of Helsinki Helsinki, Finland</i>	391,929
Adnan Darwiche <i>University of California (UCLA) Los Angeles, CA, USA</i>	101		

Tommi Junttila <i>Aalto University</i> <i>Espoo, Finland</i>	1047	Felip Manyà <i>IIIA-CSIC</i> <i>Bellaterra, Spain</i>	903
Henry Kautz <i>University of Rochester</i> <i>Rochester, NY, USA</i>	213	Vasco Manquinho <i>INESC-ID, IST – Universidade de</i> <i>Lisboa</i> <i>Lisboa, Portugal</i>	1087
Benjamin Kiesel <i>SAP Innovation Center Network</i> <i>Munich, Germany</i>	391	Paolo Marin <i>Albert Ludwig University of Freiburg</i> <i>Freiburg, Germany</i>	1157
Hans Kleine Büning <i>Paderborn University</i> <i>Paderborn, Germany</i>	571, 1131	Joao Marques-Silva <i>IRIT, CNRS</i> <i>Toulouse, France</i>	133
Daniel Kroening <i>Oxford University</i> <i>Oxford, United Kingdom</i>	791	John N. Martin <i>Professor Emeritus, University of</i> <i>Cincinnati</i> <i>Cincinnati, OH, USA</i>	3
Oliver Kullmann <i>Swansea University</i> <i>Swansea, United Kingdom</i>	351, 571	Ruben Martins <i>Carnegie Mellon University</i> <i>Pittsburgh, PA, USA</i>	929
Kevin Leyton-Brown <i>University of British Columbia</i> <i>Vancouver, Canada</i>	481	Kuldeep S. Meel <i>National University of Singapore</i> <i>Singapore</i>	1015
Chu-Min Li <i>University of Picardie Jules Verne</i> <i>Amiens, France</i>	903	Remi Monasson <i>Ecole Normale Supérieure and CNRS</i> <i>Paris, France</i>	859
Florian Lonsing <i>Stanford University</i> <i>Stanford, CA, USA</i>	1177	Massimo Narizzano <i>University of Genoa</i> <i>Genoa, Italy</i>	1157
Inês Lynce <i>Universidade de Lisboa</i> <i>Lisboa, Portugal</i>	133	Ilkka Niemelä <i>Aalto University</i> <i>Espoo, Finland</i>	1047
Hans van Maaren <i>TU Delft</i> <i>Delft, The Netherlands</i>	183	Jakob Nordström <i>University of Copenhagen /</i> <i>Lund University</i> <i>Copenhagen, Denmark /</i> <i>Lund, Sweden</i>	233
Stephen M. Majercik <i>Bowdoin College</i> <i>Brunswick, ME, USA</i>	1331	Knot Pipatsrisawat <i>University of California (UCLA)</i> <i>Los Angeles, CA, USA</i>	101
Sharad Malik <i>Princeton University</i> <i>Princeton, NJ, USA</i>	133		

Steven David Prestwich <i>University College Cork Cork, Ireland</i>	75	Guilhem Semerjian <i>PSL Research University Paris, France</i>	859
Jussi Rintanen <i>Aalto University Espoo, Finland</i>	765	Sanjit A. Seshia <i>University of California at Berkeley Berkeley, CA, USA</i>	1267
Olivier Roussel <i>Artois University Lens, France</i>	1087	Stefan Szeider <i>TU Wien Vienna, Austria</i>	693
Ashish Sabharwal <i>Allen Institute for AI Seattle, WA, USA</i>	213, 463, 993	Armando Tacchella <i>University of Genoa Genoa, Italy</i>	1223
Karem A. Sakallah <i>University of Michigan Ann Arbor, MI, USA</i>	509	Cesare Tinelli <i>The University of Iowa Iowa City, IA, USA</i>	1267
Marko Samer [†] <i>TU Wien Vienna, Austria</i>	693	Moshe Y. Vardi <i>George Professor, Rice University Houston, TX, USA</i>	1015
Roberto Sebastiani <i>University of Trento Trento, Italy</i>	859, 903	Francesco Zamponi <i>Université PSL / Sorbonne Université / Université de Paris Paris, France</i>	859
Martina Seidl <i>Johannes Kepler University Linz Linz, Austria</i>	1177	Hantao Zhang <i>The University of Iowa Iowa City, IA, USA</i>	819
Bart Selman <i>Cornell University Ithaca, NY, USA</i>	213, 993		

This page intentionally left blank