Refining restarts strategies for SAT and UNSAT

Gilles Audemard[1]⋆ and Laurent Simon[2]

[1] Univ Lille-Nord de France CRIL / CNRS UMR8188,
Lens, F-62307 audemard@cril.fr,
[2] Univ Paris-Sud, LRI / CNRS UMR8623
Orsay, F-91405 simon@lri.fr

Abstract. So-called Modern SAT solvers are built upon a few – but essential – ingredients: branching, learning, restarting and clause database cleaning. Most of them have been greatly improved since their first introduction, more than ten years ago. In many cases, the initial reasons that lead to their introduction do not explain anymore their current usage (for instance: very rapid restarts, aggressive clause database cleaning). Modern SAT solvers themselves share fewer and fewer properties with their ancestor, the classical backtrack search DPLL procedure.
In this paper, we explore restart strategies in the light of a new vision of SAT solvers. Following the successful results of GLUCOSE, we consider CDCL solvers as resolution-based producers of clauses. We show that this vision is particularly salient for targeting UNSAT formulae. In a second part, we show how detecting sudden increases in the number of variable assignments can help the solver to target SAT instances too. By varying our restart strategy, we show an important improvement over GLUCOSE 2.0, the winner of the 2011 SAT Competition, category Application SAT+UNSAT formulae. Finally we would like to point out that this new version of GLUCOSE was the winner of the SAT Challenge 2012.

1 Introduction

Despite the important progress constantly observed in the practical solving of SAT/-UNSAT problems, most of these components are commonly viewed as improvements over the DPLL-62 [4] procedure. In many works, including recent ones, the vision of CDCL (Conflict Driven Clause Learning, also called "Modern SAT") solvers [10, 13] as backtrack-search engines prevails. In this paper, we discuss this point of view and focus our work on an essential ingredient of Modern SAT solvers: restarts. Restart strategies used in CDCL solvers are also often understood as heirs of the DPLL framework. If the search seems to not be close to finding a model, then a restart often means "restarts tree-search elsewhere".

Restarting was first proposed to prevent the search tree from making mistakes in its top decisions. This idea was formally observed by the *Heavy Tailed* distribution of CPU time of DPLL-like solvers. However, initially, restarting was thought of as a witness of branching heuristics failure. Thus, restarts were not (*i.e.* after thousands of explored branches) and, to ensure completeness of approaches, restarts were triggered according

⋆ This work has been partially supported by CNRS and OSEO, under the ISI project "Pajero".

to a geometric series. More recently, restarts strategies were following Luby series, a linear increasing series.

In this work, we take an alternate vision of CDCL solvers. Our goal is to view them as resolution-based "clauses producers" instead of "search-space explorers". This vision is illustrated by our solver GLUCOSE. It was already embedded in GLUCOSE 1.0 [1], thanks to a good measure over learnt clauses, called LBD (Literal Block Distance), however, some details of the restart strategy used in the version of GLUCOSE that participated to the SAT Competition 2009 [3] [8] were never published. Additionally, in this paper, we describe a new restart strategy, used in GLUCOSE 2.1, that shows significant improvements over GLUCOSE 2.0.[4] This new restart strategy is twice as aggressive and still does not have any guarantee that long runs will be allowed. This property, associated with the fact that GLUCOSE 2.0 deleted 93% of the learnt clauses during the SAT competition 2011,[5] shows that GLUCOSE can hardly be considered as a systematic search solver, and is increasingly distant from the DPLL vision of Modern SAT Solvers. We also show in this paper that such an aggressive strategy is especially good for UNSAT problems. In order to allow an improvement also on SAT instances, we propose to add a "blocking" parameter to our strategy, which is able to postpone one or many aggressive restarts when the solver seems to suddenly reach a global assignment. Intuitively, our work tries to reconcile the two opposing forces working in CDCL solvers. One (the UNSAT force) is trying to decrease the number of decision levels, by learning good clauses. The other (the SAT force) is trying to increase the number of decision levels, by assigning more and more variables.

2 Background

For lack of space, we assume the reader is familiar with Satisfiability notions (variables x_i, literal x_i or $\neg x_i$, clause, unit clause and so on). We just recall the global schema of CDCL solvers [5, 10]: a typical branch can be seen as a sequence of decisions (usually with the VSIDS heuristic) followed by propagations, repeated until a conflict is reached. Each decision literal is assigned at its own level shared with all propagated literals assigned at the same level (a block is defined as all literals which are assigned at the same decision level). Each time a conflict is reached, a *nogood* is extracted. The learnt clause is then added to the clause database and a *backjumping* level is computed from it. Solvers also incorporate other components such as restarts (see section 3) and a learnt clause database reduction policy.

This last component is clearly crucial to solver performance. Indeed, keeping too many learnt clauses will slow down the unit propagation process, while deleting too many of them will break the overall learning benefit. Then, more or less frequently, half of the learnt clauses are removed from the database. Consequently, identifying good learnt clauses - relevant to the proof derivation - is clearly an important challenge.

[3] GLUCOSE 1.0 won the first prize in the category Application UNSAT, SAT Competition 2009

[4] GLUCOSE 2.0 won the first prize in the category Application SAT+UNSAT, 2011

[5] If we study all GLUCOSE's traces of the 2011 competition, phase 2, in the categories Applications and Crafted, GLUCOSE 2.0 learnt 973,468,489 clauses (sum over all traces) but removed 909,123,525 of them during computation, i.e. more than 93end.

The first proposed quality measure follows the success of the activity based VSIDS heuristic. More precisely, a learnt clause is considered relevant to the proof, if it is involved more often in recent conflicts, *i.e.* usually used to derive asserting clauses. Clearly, this deletion strategy is based on the assumption that a useful clause in the past could be useful in the future. More recently, a more accurate measure called LBD (*Literal Block Distance*) was proposed to estimate the quality of a learnt clause. This measure is defined as follow [1]:

Definition 1 (Literal Block Distance (LBD)). *Given a clause c, and a partition of its literals into n subsets according to the current assignment, s.t. literals are partitioned w.r.t their level. The LBD of C is exactly n.*

Extensive experiments demonstrated that clauses with small LBD values are used more often (in propagation and conflict) than those with higher LBD values [1]. Intuitively, the LBD score measures the number of decisions that a given clause links together. It was shown that "glue clauses", *i.e.*, clauses of LBD 2, are very important. Then, this new measure can be combined with a very agressive cleaning strategy: bad clauses are often deleted.

Another important component for our purposes is the literal polarity to be chosen when the next decision variable is selected, by the VSIDS heuristic. Usually, a default polarity (e.g. false) is defined and used each time a decision literal is assigned. Based on the observation that restarting and backjumping might lead to repetitive solving the same subformulas, Pipatsrisawat and Darwiche [12] proposed to dynamically save for each variable the last seen polarity. This literal polarity based heuristic, called phase saving, prevents the solver from solving the same satisfiable subformulas several times. As we will see in the next section, this component is crucial for rapid restarts.

3 State of the (Rest)Art

In CDCL solvers, restarting is not restarting. The solver maintains all its heuristic values between them and restarts must be seen as dynamic rearrangements of variable dependencies. Additionally, hidden restarts are triggered in all solvers: when a unit clause is learnt (which is often the case in many problems, during the first conflicts), the solver usually cancels all its decisions without trying to recover them, in order to immediately consider this new fact. Thus, on problems with a lot of learnt unit clauses, the solver will restart more often, but silently.

There has been substantial previous work in this area. We only review here the most significant ones for our own purposes. In 2000 [6], the heavy-tailed distribution of backtracking (DPLL) procedures was identified. To avoid the high variance of the runtime and improve the average performance of SAT solvers, it was already proposed to add restarts to DPLL searches. As we will see, this explanation does not hold anymore for explaining the performance of recent solvers.

In MINISAT 1.4 [5], a scheduled restart policy was incorporated, following a geometric series (common ratio of 1.5, starting after 100 conflicts). In [7] the Luby series [9] was demonstrated to be very efficient. This series follows a slow-but-exponentially increasing law like 1 1 2 1 1 2 4 1 1 2 4 8 1 1 2 1 ... Of course, in practice, this series is multiplied by a factor (in general between 64 and 256). This series is interesting because

		X						
K	SAT?	50	75	100	200	300	400	500
0.7	N	**89**	86	86	81	85	75	80
0.8	N	**93**	**93**	90	86	85	87	82
0.7	Y	78	81	78	77	81	77	**83**
0.8	Y	78	77	76	**79**	76	78	74

Table 1. Number of solved instances with different values of X (size of the `queueLBD`). Results are refined by category of benchmarks (SAT, UNSAT) and the margin ratio (0.7 or 0.8). Tests were (here) conducted on SAT 2011 Application benchmarks only. The CPU CutOff was 900s.

it was proven to be optimal for search algorithms in which individual runs are independent, i.e., share no information. We will discuss the problems of this restart strategy later.

It was proposed in [3] to nest two series, a geometric one and a Luby. The idea was to ensure that restarts are guaranteed to increase (allowing the search to, theoretically, be able to terminate) and that fast restarts occur more often than in the geometric series. In [2], it was proposed to postpone any scheduled restart by observing the "agility" of the solver. This measure is based on the polarity of the phase saving mechanism (see section 2). If most of the variables are forced against their saved polarity, then the restart is postponed: the solver might find a refutation soon. If polarities are stalling, the scheduled restart is triggered.

We should probably note here that the phase saving scheme [12] is crucial in the case of fast restarts. Indeed, it allows the solver to recover most of the decisions (but not necessarily in the same order) before and after the restart.

3.1 A note on the Luby restart strategy

Luby restarts were shown to be very efficient even for very small values (a factor of 6 was shown to be optimal [7] in terms of number of conflicts). However, in most of the CDCL solvers using Luby restarts, the CPU time needs to be taken into account and a typical Luby factor is often between 32 and 256.

To have an intuition of the Luby behavior, if we take a constant ratio of 50 (to compare it with GLUCOSE 2.1 that will be presented in the following section), after 1,228,800 conflicts, the CDCL solver will have triggered 4095 restarts, with an average of 300 conflicts per restart and one long run of 102,400 conflicts (and two runs of 51,200 conflicts, four runs of 25,600 conflicts, ...). Thus, if a few top-level decisions of the longest run were wrong, 8% of the total effort will be lost. Considering the fact that this longest run is triggered without any special "stronger" heuristic (one can branch on very recent variables only for instance, leading this long run to a very local sub-problem), this is a risky strategy. Clearly, there is room for improvements.

4 Faster and More Reactive Restarts for UNSAT

In the first version of GLUCOSE we experimented with a restart policy based on the decrease of decision levels. The aim was to encourage the solver to minimize the number

of decisions before reaching conflicts. We showed that, on many instances, the faster the decreasing is, the better the performance (this observation was the initial motivation for GLUCOSE). However, even if this strategy was used in [1], additional experiments quickly pointed out the importance of good clauses (having a low LBD score). The overall architecture of GLUCOSE was quickly fully-oriented to focus on LBD scores only, so it was natural to rely on the production of good clauses instead of the production of fewer decision levels as possible (this last measure was shown to be stalling after a while, or even constantly increasing on some particular benchmarks. Taking LBD scores only into account was shown to be more informative). Thus, we decided to change the restart policy of the version of GLUCOSE that participated to the SAT 2009 competition.

The idea behind our restart strategy is the following: since we want good clauses (w.r.t. LBD), we perform a restart if the last produced ones have high LBDs. To do that, if K times ($0 < K < 1$) the average of LBD scores of the last X conflicts was above the average of all LBD scores of produced clauses so far, then a restart is triggered. To be able to compute the moving average of the X last LBDs we use a bounded queue (called `queueLBD`) of size X. Of course, whenever the bounded queue is not full, we can not compute the moving average, then at least X conflicts are performed before making a restart. We produce here a sketch of the algorithm that triggers restarts.

```
// In case of conflict
compute learnt clause c;
sumLBD+= c.lbd(); conflicts++;// Used for global average
queueLBD.push(c.lbd());
if(queueLBD.isFull() && queueLBD.avg()*K>sumLBD/conflicts) {
     queueLBD.clear();
     restart();
}
```

The magic constant K, called the *margin ratio*, provides different behaviors. The larger K is, the fewer restarts are performed. In the first version of GLUCOSE, we use $X = 100$ and $K = 0.7$. These values were experimentally fixed to give good results on both SAT and UNSAT problems. This strategy was not changed in GLUCOSE 2.0.

We show now how we can improve the value of X by observing the performance of GLUCOSE on SAT/UNSAT problems. We must also say, as a preliminary, that improving SAT solvers is often a cruel world. To give an idea, improving a solver by solving at least ten more instances (on a fixed set of benchmarks of a competition) is generally showing a critical new feature. In general, the winner of a competition is decided based on a couple of additional solved benchmarks.

Table 1 provides for two margin ratios (0.7 and 0.8, chosen for their good performance), the number of solved instances from the SAT 2011 competition when changing the size of the bounded queue (X). We also need to clarify that 0.7 was used in GLUCOSE 1.0 and 2.0, while 0.8 was, also, already proposed in GLUEMINISAT [11]. As we can see, the size X of the bounded queue has a major impact on UNSAT formulae, and not much effect on SAT instances: for both values of K, the number of solved UNSAT instances are clearly decreasing, while the SAT ones are stalling. Thus, bounded queue size of 50 seems to be a very good value for UNSAT problems. In order to understand the impact of changing the bounded queue size, let us notice here that fixing X has not only the effect of restarting more or less frequently (this is a strict minimal interval

length for restarts: no restart is triggered while the bounded queue is still not full), it has also an important side effect: the longer the queue is, the slower it can react to important but short variations. Large average windows can "absorb" short but high variances. Thus, a small value for X will have a more reactive behavior, leading to even more restarts.

As a short conclusion for this section, we see that, by assigning more reactive values to the restart strategy of GLUCOSE 1.0, GLUEMINISAT used very good parameters for UNSAT problems. This observation certainly explain why GLUEMINISAT won a first prize in the UNSAT category in 2011. Next, we show how we can keep these parameters while also allowing more SAT instances to be solved.

5 Blocking a fast restart when approaching a full assignment

As mentioned above, learning good clauses should allow the solver to make fewer and fewer decisions before reaching conflicts. However, we have clearly shown that reducing the bounded queue size has no outstanding effect on SAT instances. In this section, we show how we can add a new blocking strategy for postponing restarts, when the solver is approaching a global solution in order to improve our solver on SAT instances.

The problem we have is the following. GLUCOSE is firing aggressive clause deletion (see section 2) and fast restarts. On some instances, restarts are really triggered every 50 conflicts. So, if the solver is trying to reach a global assignment, it has now only a few tries before reaching it. Additionally, the aggressive clause deletion strategy may have deleted a few clauses that are needed to reach the global assignment directly. The idea we present here is to simply delay for one turn the next restart (the bounded queue is emptied and the restart possibility will be tested only when it is full again) each time the number of total assignments are significantly above the average measured during a window of last conflicts (we chose a rather large window of 5,000 conflicts).

The total number of assignments (called trail size) is the current number of decisions plus the number of propagated literals. Let us now formalize the notion of "significantly above" expressed above. We need for this an additional margin value, that we will call R. Our idea is to empty the bounded queue of LBD (see section 4), thus postponing a possible restart, each time a conflict is reached with a trail size greater than R times the moving average of the last 5,000 conflicts trail sizes (computed using an other bounded queue called `trailQueue`). Of course, this can occur many times during an interval of restarts, and thus the bounded queue can be emptied before it is again full, or can be emptied many times during the interval. We produce here the sketch of the algorithm that blocks restarts.

```
// In case of conflict
queueTrail.push(trail.size());
if(queueLBD.isFull() && queueTrail.isFull() &&trail.size()>R*queueTrail.avg()) {
      queueLBD.clear();
}
```

Table 2 helps us determine the right value for R, according to the different choices of (X,K): ($(100, 0.7)$ and $(50, 0.8)$) Here are a few conclusions that can be drawn from this Table:

- For UNSAT problems, we observe that the R value does not play any significant role. If we take a look on the average number of conflicts between each restart

		R				
(X, K)	SAT?	No	1.2	1.3	1.4	1.5
$(100, 0.7)$	N	**86** (4500)	85 (4700)	**86** (3600)	83 (4600)	**86** (4400)
$(50, 0.8)$	N	93 (318)	89 (364)	93 (375)	**94** (369)	93 (400)
$(100, 0.7)$	Y	78 (8200)	77 (7700)	79 (7500)	76 (8600)	**80** (12000)
$(50, 0.8)$	Y	78 (433)	80 (710)	78 (750)	**87** (520)	83 (561)

Table 2. Number of solved instances with different values of R with, in parenthesis, the average number of conflicts between two restarts ("No" means no blocking). Tests were conducted on SAT 2011 Application benchmarks only. The CPU CutOff was 900s.

(average over all solved instances) we can notice that the size of the bounded queue increases a lot the number of conflicts between each restart. Here, however, the blocking strategy has no real impact.

– For SAT problems, we observe a few interesting things. First, varying the value of the R parameter does not impact performance when $(X, K) = (100, 0.7)$ (we also observed this on larger values than 100). Second, for the couple $(50, 0.8)$, blocking restarts seems very promising. Restarts are very aggressive and the blocking strategy it may be crucial to postpone a restart. Setting R to 1.4 is clearly the best choice. The results improve the number of solved SAT instances by solving 9 additional SAT benchmarks. We can also notice that, (1) the different values of R have an impact on the average number of conflicts between two restarts and, (2) in case of SAT instances the average number of conflicts is always greater than for UNSAT ones.

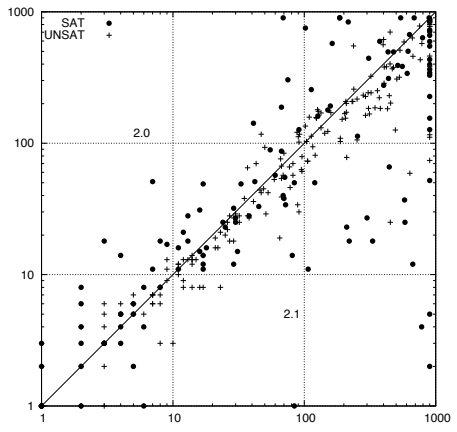
As a conclusion here, we experimentally show that playing on the R value has a great impact on SAT instances, and almost no impact on UNSAT ones. This can be observed by looking at the average number of conflicts between restarts. $R = 1.4$ allows larger windows for SAT while maintaining small ones for UNSAT.

6 Experimental evaluation of GLUCOSE 2.1

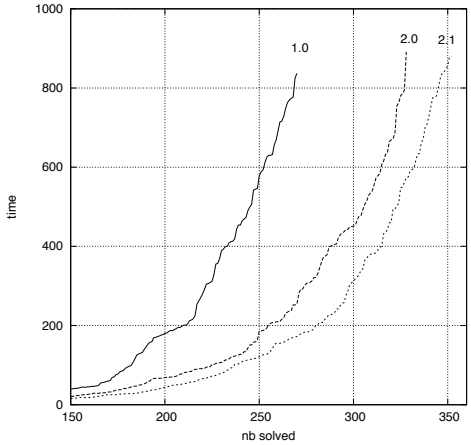
Progresses made in the practical solving of SAT instances are constantly observed and important, even in the last few years. We illustrate here the progress made since our first release of GLUCOSE 1.0, 3 years ago. To understand the progress made, we must also notice that GLUCOSE 2.0 is based on MINISAT 2.2 (instead of MINISAT 2).

Our methodology was the following. We worked on GLUCOSE 2.1 on 2011 benchmarks only, then tested our final ideas on all benchmarks (2009+2011 benchmarks). Our aim was somehow to counterbalance the possibility that GLUCOSE 2.1 would have been specialized for 2011 benchmarks (the 2.0 version had no access to them). As we will see, we also measured an important improvement on benchmarks from the 2009 competition.

First of all, the scatter plot in Figure 1(a) provides a graphical comparison of version 2.0 and 2.1 on all instances. Since most dots are below the diagonal, it is clear that version 2.1 is faster than 2.0. We also see on the traditional cactus plot shown in Figure 1(b) the breakthrough improvements made in the last 3 years. The novelty of



(a) GLUCOSE 2.1 VS GLUCOSE 2.0



(b) Cactus plot

		1.0			2.0			2.1		
Benchmarks	nb	SAT	UNSAT	Total	SAT	UNSAT	Total	SAT	UNSAT	Total
SAT2009	292	50	89	139	68	118	186	**77**	**123**	**200**
SAT2011	300	74	79	153	78	86	164	**87**	**94**	**181**
SAT2009+2011	536	113	157	270	136	192	328	**148**	**204**	**352**

(c) Number of solved benchmarks in different competitions

Fig. 1. a. Scatter plot: Each dot represents an instances, x-axis (resp. y-axis) the time needed by version 2.0 (resp. 2.1) to solve it. Dots below the diagonal represent instances solved faster with version 2.1 (log scale).
b. Cactus plot of 3 versions: The x-axis gives the number of solved instances and the y-axis the time needed to solve them.
c. For each competition, the number of benchmarks (nb) is provided with, for each versions of GLUCOSE, the number of solved instances.

GLUCOSE 2.1 vs GLUCOSE 2.0 is less important, but still significant and relies only on the improvements presented in this paper.

Finally, Figure 1(c) details the obtained results on the 2 sets of benchmarks, with the same CPU CutOff of 900 (of course benchmarks common to both sets are reported only one time in last line). Clearly, the new version of GLUCOSE, based on more aggressive restarts presented in this paper, obtained the best results.

7 Conclusion

In this paper we show how, by refining the dynamic strategy of GLUCOSE for UNSAT problems, and adding a new and simple blocking strategy to it, that is specialized for SAT problems, we are able to solve significantly more problems, more quickly. The overall idea of this paper is also to push a new vision of CDCL SAT solvers. We think they may now be closer to resolution-based producers of good clauses rather than backtrack search engines.

References

1. Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *proceedings of IJCAI*, pages 399–404, 2009.
2. 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, pages 28–33, Berlin, Heidelberg, 2008. Springer-Verlag.
3. Armin Biere. Picosat essentials. *JSAT*, 4:75–97, 2008.
4. Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communication of ACM*, 5(7):394–397, 1962.
5. Niklas Een and Niklas Sörensson. An extensible SAT-solver. In *proceedings of SAT*, pages 502–518, 2003.
6. Carla Gomes, Bart Selman, Nuno Crato, and Henry Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of automated reasoning*, 24:67–100, 2000.
7. Jimbo Huang. The effect of restarts on the efficiency of clause learning. In *proceedings of IJCAI*, pages 2318–2323, 2007.
8. Daniel Le Berre, Matti Jarvisalo, Olivier Roussel, and Laurent Simon. SAT competitions, 2002–2011. http://www.satcompetition.org/.
9. Mickael Luby, Allistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. In *Israel Symposium on Theory of Computing Systems*, pages 128–133, 1993.
10. Matthew Moskewicz, Connor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *proceedings of DAC*, pages 530–535, 2001.
11. Hidetomo Nabeshima, Koji Iwanuma, and Katsumi Inoue. Glueminisat2.2.5. System Desciption, available on glueminisat.nabelab.org.
12. Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *proceedings of SAT*, pages 294–299, 2007.
13. Joao Marques Silva and Karem Sakallah. Grasp - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.