Learning Rate Based Branching Heuristic for SAT Solvers

Jia Hui Liang(✉), Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki

University of Waterloo, Waterloo, Canada
jliang@gsd.uwaterloo.ca

Abstract. In this paper, we propose a framework for viewing solver branching heuristics as optimization algorithms where the objective is to maximize the *learning rate*, defined as *the propensity for variables to generate learnt clauses*. By viewing online variable selection in SAT solvers as an optimization problem, we can leverage a wide variety of optimization algorithms, especially from machine learning, to design effective branching heuristics. In particular, we model the variable selection optimization problem as an online multi-armed bandit, a special-case of *reinforcement learning*, to learn branching variables such that the learning rate of the solver is maximized. We develop a branching heuristic that we call *learning rate branching* or LRB, based on a well-known multi-armed bandit algorithm called *exponential recency weighted average* and implement it as part of MiniSat and CryptoMiniSat. We upgrade the LRB technique with two additional novel ideas to improve the learning rate by accounting for *reason side rate* and exploiting *locality*. The resulting LRB branching heuristic is shown to be faster than the VSIDS and conflict history-based (CHB) branching heuristics on 1975 application and hard combinatorial instances from 2009 to 2014 SAT Competitions. We also show that CryptoMiniSat with LRB solves more instances than the one with VSIDS. These experiments show that LRB improves on state-of-the-art.

1 Introduction

Modern Boolean SAT solvers are a critical component of many innovative techniques in security, software engineering, hardware verification, and AI such as solver-based automated testing with symbolic execution [9], bounded model checking [11] for software and hardware verification, and planning in AI [27] respectively. Conflict-driven clause-learning (CDCL) SAT solvers [4,6,12,23, 24,29] in particular have made these techniques feasible as a consequence of their surprising efficacy at solving large classes of real-world Boolean formulas. The development of various heuristics, notably the Variable State Independent Decaying Sum (VSIDS) [24] branching heuristic (and its variants) and conflict analysis techniques [23], have dramatically pushed the limits of CDCL solver performance. The VSIDS heuristic is used in the most competitive CDCL SAT solvers such as Glucose [4], Lingeling [6], and CryptoMiniSat [29]. Since its

N. Creignou and D. Le Berre (Eds.): SAT 2016, LNCS 9710, pp. 123–140, 2016.
DOI: 10.1007/978-3-319-40970-2_9

introduction in 2001, VSIDS has remained one of the most effective and dominant branching heuristic despite intensive efforts by many researchers to replace it [7,15,16,28]. In early 2016, we provided the first branching heuristic that is more effective than VSIDS called the conflict history-based (CHB) branching heuristic [19]. The branching heuristic introduced in this paper, which we refer to as *learning rate branching* (LRB), significantly outperforms CHB and VSIDS.

In this paper, we introduce a general principle for designing branching heuristics wherein online variable selection in SAT solvers is viewed as an optimization problem. The objective to be maximized is called the *learning rate* (LR), a numerical characterization of a variable's propensity to generate learnt clauses. The goal of the branching heuristic, given this perspective, is to select branching variables that will maximize the cumulative LR during the run of the solver. Intuitively, achieving a perfect LR of 1 implies the assigned variable is responsible for every learnt clause generated during its lifetime on the assignment trail.

We put this principle into practice in this paper. Although there are many algorithms for solving optimization problems, we show that *multi-armed bandit learning* (MAB) [31], a special-case of *reinforcement learning*, is particularly effective in our context of selecting branching variables. In MAB, an agent selects from a set of actions to receive a reward. The goal of the agent is to maximize the cumulative rewards received through the selection of actions. As we will describe in more details later, we abstract the branching heuristic as the agent, the available branching variables are abstracted as the actions, and LR is defined to be the reward. Abstracting online variable selection as a MAB problem provides the bridge to apply MAB algorithms from the literature directly as branching heuristics. In our experiments, we show that the MAB algorithm called *exponential recency weighted average* (ERWA) [31] in our abstraction surpasses the VSIDS and CHB branching heuristics at solving the benchmarks from the 4 most recent SAT Competitions in an apple-to-apple comparison. Additionally, we provide two extensions to ERWA that increases its ability to maximize LR and its performance as a branching heuristic. The final branching heuristic, called *learning rate branching* (LRB), is shown to dramatically outperform CryptoMiniSat [29] with VSIDS.

1.1 Contributions

Contribution I: We define a principle for designing branching heuristics, that is, a branching heuristic should maximize the *learning rate* (LR). We show that this principle yields highly competitive branching heuristics in practice.

Contribution II: We show how to abstract online variable selection in the *multi-armed bandit* (MAB) framework. This abstraction provides an interface for applying MAB algorithms directly as branching heuristics. Previously, we developed the *conflict history-based* (CHB) branching heuristic [19], also inspired by MAB. The key difference between this paper and CHB is that in the case of CHB the rewards are known a priori, and there is no metric being optimized. Whereas in this work, the learning rate is being maximized and is unknown a priori, which requires a bona fide machine learning algorithm to optimize under uncertainty.

Contribution III: We use the MAB abstraction to develop a new branching heuristic called *learning rate branching* (LRB). The heuristic is built on a well-known MAB algorithm called *exponential recency weighted average* (ERWA). Given our domain knowledge of SAT solving, we extend ERWA to take advantage of *reason side rate* and *locality* [20] to further maximize the learning rate objective. We show in comprehensive apple-to-apple experiments that it outperforms the current state-of-the-art VSIDS [24] and CHB [19] branching heuristics on 1975 instances from four recent SAT Competition benchmarks from 2009 to 2014 on the application and hard combinatorial categories. Additionally, we show that a modified version of CryptoMiniSat with LRB outperforms Glucose, and is very close to matching Lingeling over the same set of 1975 instances.

2 Preliminaries

2.1 Simple Average and Exponential Moving Average

Given a time series of numbers $\langle r_1, r_1, r_2, ..., r_n \rangle$, the *simple average* is computed as $avg(\langle r_1, ..., r_n \rangle) = \sum_{i=1}^{n} \frac{1}{n} r_i$. Note that every r_i is given the same coefficient (also called *weight*) of $\frac{1}{n}$.

In a time series however, recent data is more pertinent to the current situation than old data. For example, consider a time series of the price of a stock. The price of the stock from yesterday is more correlated with today's price than the price of the stock from a year ago. The *exponential moving average* (EMA) [8] follows this intuition by giving the recent data higher weights than past data when averaging. Incidentally, the same intuition is built into the multiplicative decay in VSIDS [5,20]. The EMA is computed as $ema_\alpha(\langle r_1, ..., r_n \rangle) = \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} r_i$ where $0 < \alpha < 1$ is called the *step-size* parameter. α controls the relative weights between recent and past data. EMA can be computed incrementally as $ema_\alpha(\langle r_1, ..., r_n \rangle) = (1-\alpha) \cdot ema_\alpha(\langle r_1, ..., r_{n-1} \rangle) + \alpha r_n$, and we define the base case $ema_\alpha(\langle \rangle) = 0$.

2.2 Multi-Armed Bandit (MAB)

We will explain the MAB problem [31] through a classical analogy. Consider a gambler in a casino with n slot machines, where the objective of the gambler is to maximize payouts received from these machines. Each slot machine has a probability distribution describing its payouts, associating a probability with every possible value of payout. This distribution is hidden from the gambler. At any given point in time, the gambler can play one of the n slot machines, and hence has n actions to choose from. The gambler picks an action, plays the chosen slot machine, and receives a *reward* in terms of monetary payout by sampling that slot machine's payout probability distribution. The MAB problem is to decide which actions to take that will maximize the cumulative payouts.

If the probability distributions of the slot machines were revealed, then the gambler would simply play the slot machine whose payout distribution has the highest mean. This will maximize expected payouts for the gambler. Since the probability distribution is hidden, a simple MAB algorithm called *sample-average* [31] estimates the true mean of each distribution by averaging the samples of observed payouts. For example, suppose there are 2 slot machines. The gambler plays the first and the second slot machine 4 times each, receiving the 4 payouts $\langle 1, 2, 3, 4 \rangle$ and $\langle 5, 4, 3, 2 \rangle$ respectively. Then the algorithm will estimate the mean of the first and second slot machines' payout distributions as $avg(\langle 1, 2, 3, 4 \rangle) = 2.5$ and $avg(\langle 5, 4, 3, 2 \rangle) = 3.5$ respectively. Since the second slot machine has a higher estimated mean, the choice is to play the second slot machine. This choice is called *greedy*, that is, it chose the action it estimates to be the best given extant observations. On the other hand, choosing a non-greedy action is called *exploration* [31].

The sample-average algorithm is applicable if the hidden probability distributions are fixed. If the distributions change over time, then the problem is called *nonstationary*, and requires different algorithms. For example, suppose a slot machine gives smaller and smaller payouts the more it has been played. The older the observed payout, the bigger the difference between the current probability distribution and the distribution from which the payout was sampled. Hence, older observed payouts should have smaller weights. This gives rise to the *exponential recency weighted average* [31] (ERWA) algorithm. Instead of computing the simple average of the observed payouts, use EMA to give higher weights to recent observations relative to distant observations. Continuing the prior example, ERWA estimates the mean payout of the first and second slot machines as $ema_\alpha(\langle 1, 2, 3, 4 \rangle) = 3.0625$ and $ema_\alpha(\langle 5, 4, 3, 2 \rangle) = 2.5625$ respectively where $\alpha = 0.5$. Therefore ERWA estimates the first slot machine to have a higher mean, and hence the greedy action is to play the first slot machine.

2.3 Clause Learning

The defining feature of CDCL solvers is to analyze every conflict it encounters to learn new clauses to block the same conflicts, and up to exponentially similar conflicts, from re-occurring. The solver maintains an implication graph, a directed acyclic graph where the vertices are assigned variables and edges record the propagations between variables induced by Boolean constraint propagation. A clause is falsified when all of its literals are assigned to false, and in this circumstance, the solver can no longer proceed with the current assignment. The solver analyzes the implication graph and *cuts* the graph into two sides: the *conflict side* and the *reason side*. The conflict side must contain all the variables from the falsified clause and the reason side must contain all the decision variables. A learnt clause is generated on the variables from the reason side incident to the cut by negating the current assignments to those variables. In practice, the implication graph is typically cut at the first unique implication point [33]. Upon learning a clause, the solver backtracks to an earlier state where no clauses are falsified and proceeds from there.

2.4 The VSIDS Branching Heuristic

VSIDS can be seen as a ranking function that maintains a floating point number for each Boolean variable in the input formula, often called activity. The activities are modified in two interweaving operations called the bump and the multiplicative decay. Bump increases the activity of a variable additively by 1 whenever it appears in either a newly learnt clause or the conflict side of the implication graph. Decay periodically decreases the activity of every variable by multiplying all activities by the decay factor δ where $0 < \delta < 1$. Decay typically occurs after every conflict. VSIDS ranks variables in decreasing order of activity, and selects the unassigned variable with the highest activity to branch on next. This variable is called the *decision* variable. A separate heuristic, typically phase-saving [26], will select the polarity to assign the decision variable.

3 Contribution I: Branching Heuristic as Learning Rate (LR) Optimization

The branching heuristic is responsible for assigning variables through decisions that the SAT solver makes during a run. Although most of the assignments will eventually revert due to backtracking and restarts, the solver guarantees progress due to the production of learnt clauses. It is well-known that branching heuristics play a significant role in the performance of SAT solvers. To frame branching as an optimization problem, we need a metric to quantify the degree of contribution from an assigned variable to the progress of the solver, to serve as an objective to maximize. Since producing learnt clauses is a direct indication of progress, we define our metric to be the variable's propensity to produce learnt clauses. We will now define this formally.

Clauses are learnt via conflict analysis on the implication graph that the solver constructs during solving. A variable v *participates* in generating a learnt clause l if either v appears in l or v is resolved during the conflict analysis that produces l (i.e., appears in the conflict side of the implication graph induced by the cut that generates l). In other words, v is required for the learning of l from the encountered conflict. Note that only assigned variables can participate in generating learnt clauses. We define I as the interval of time between the assignment of v until v transitions back to being unassigned. Let $P(v, I)$ be the number learnt clauses in which v participates during interval I and let $L(I)$ be the number of learnt clauses generated in interval I. The *learning rate* (LR) of variable v at interval I is defined as $\frac{P(v,I)}{L(I)}$. For example, suppose variable v is assigned by the branching heuristic after 100 learnt clauses are produced. It participates in producing the 101-st and 104-th learnt clause. Then v is unassigned after the 105-th learnt clause is produced. In this case, $P(v, I) = 2$ and $L(I) = 5$ and hence the LR of variable v is $\frac{2}{5}$.

The exact LR of a variable is usually unknown during branching. In the previous example, variable v was picked by the branching heuristic after 100 learnt clauses are produced, but the LR is not known until after the 105-th learnt clause is produced. Therefore optimizing LR involves a degree of uncertainty, which makes the problem well-suited for learning algorithms. In addition, the LR of a variable changes over time due to modifications to the learnt clause database, stored phases, and assignment trail. As such, estimating LR requires nonstationary algorithms to deal with changes in the underlying environment.

4 Contribution II: Abstracting Online Variable Selection as a Multi-Armed Bandit (MAB) Problem

Given n Boolean variables, we will abstract online variable selection as an n-armed bandit optimization problem. A branching heuristic has n actions to choose from, corresponding to branching on any of the n Boolean variables. The expressions *assigning a variable* and *playing an action* will be used interchangeably. When a variable v is assigned, then v can begin to participate in generating learnt clauses. When v becomes unassigned, the LR r is computed and returned as the reward for playing the action v. The terms *reward* and *LR* will be used interchangeably. The MAB algorithm uses the reward to update its internal estimates of the action that will maximize the rewards.

The MAB algorithm is limited to picking actions corresponding to unassigned variables, as the branching heuristic can only branch on unassigned variables. This limitation forces some exploration, as the MAB algorithm cannot select the same action again until the corresponding variable is unassigned due to backtracking or restarting. Although the branching heuristic is only assigning one variable at a time, it indirectly assigns many other variables through propagation. We include the propagated variables, along with the branched variables, as plays in the MAB framework. That is, branched and propagated variables will all receive their own individual rewards corresponding to their LR, and the MAB algorithm will use all these rewards to update its internal estimates. This also forces some exploration since a variable ranked poorly by the MAB algorithm can still be played through propagation.

5 Contribution III: Learning Rate Branching (LRB) Heuristic

Given the MAB abstraction, we first use the well-known ERWA bandit algorithm as a branching heuristic. We will upgrade ERWA with two novel extensions to arrive at the final branching heuristic called the *learning rate branching* (LRB) heuristic. We will justify these extensions experimentally through the lens of MAB, that is, these extensions are better at maximizing the LR rewards. We will demonstrate empirically the effectiveness of LRB at solving the benchmarks from the 4 previous SAT Competitions.

5.1 Exponential Recency Weighted Average (ERWA)

We will explain how to apply ERWA as a branching heuristic through the MAB abstraction. First we will provide a conceptual explanation, that is easier to comprehend. Then we will provide a complementary explanation from the implementation's perspective, which is equivalent to the conceptual explanation, but provides more details.

Conceptually, each variable v maintains its own time series ts_v containing the observed rewards for v. Whenever a variable v transitions from assigned to unassigned, ERWA will calculate the LR r for v (see Sect. 3) and append the reward r to the time series by updating $ts_v \leftarrow append(ts_v, r)$. When the solver requests the next branching variable, ERWA will select the variable v^* where $v^* = argmax_{v \in U}(ema_\alpha(ts_v))$ and U is the set of currently unassigned variables.

The actual implementation takes advantage of the incrementality of EMA to avoid storing the time series ts, see Algorithm 1 for pseudocode of the implementation. Alternative to the above description, each variable v maintains a floating point number Q_v representing $ema_\alpha(ts_v)$. When v receives reward r, then the implementation updates Q_v using the incrementality of EMA, that is, $Q_v \leftarrow (1 - \alpha) \cdot Q_v + \alpha \cdot r$ (see line 24 of Algorithm 1). When the solver requests the next branching variable, the implementation will select the variable v^* where $v^* = argmax_{v \in U} Q_v$ and U is the set of currently unassigned variables (see line 28 of Algorithm 1). Note that Q_v can be stored in a priority queue for all unassigned variables v, hence finding the maximum will take logarithmic time in the worst-case. The implementation is equivalent to the prior conceptual description, but significantly more efficient in both memory and time.

For our experiments, we initialize the step-size $\alpha = 0.4$. We follow the convention of typical ERWA to decrease the step-size over time [31]. After each conflict, the step-size is decreased by 10^{-6} until it reaches 0.06 (see line 14 in Algorithm 1), and remains at 0.06 for the remainder of the run. This step-size management is equivalent to the one in CHB [19] and is similar to how the Glucose solver manages the VSIDS decay factor by increasing it over time [4].

5.2 Extension: Reason Side Rate (RSR)

Recall that LR measures the participation rate of variables in generating learnt clauses. That is, variables with high LR are the ones that frequently appear in the generated learnt clause and/or the conflict side of the implication graph. If a variable appears on the reason side near the learnt clause, then these variables just missed the mark. We show that accounting for these close proximity variables, in conjunction with the ERWA heuristic, optimizes the LR further.

More precisely, if a variable v appears in a reason clause of a variable in a learnt clause l, but does not occur in l, then we say that v *reasons* in generating the learnt clause l. We define I as the interval of time between the assignment of v until v transitions back to being unassigned. Let $A(v, I)$ be the number of learnt clauses which v reasons in generating in interval I and let $L(I)$ be the

Algorithm 1. Pseudocode for ERWA as a branching heuristic using our MAB abstraction for maximizing LR.

```
procedure INITIALIZE                                ▷ Called once at the start of the solver.
    $\alpha \leftarrow 0.4$                           ▷ The step-size.
    $LearntCounter \leftarrow 0$                      ▷ The number of learnt clauses generated by the solver.
    for $v \in Vars$ do                               ▷ $Vars$ is the set of Boolean variables in the input CNF.
        $Q_v \leftarrow 0$                            ▷ The EMA estimate of $v$.
        $Assigned_v \leftarrow 0$                     ▷ When $v$ was last assigned.
        $Participated_v \leftarrow 0$                 ▷ The number of learnt clauses $v$ participated in generating since $Assigned_v$.

procedure AFTERCONFLICTANALYSIS($learntClauseVars \subseteq Vars$, $conflictSide \subseteq Vars$)    ▷ Called after a learnt clause is generated from conflict analysis.
    $LearntCounter \leftarrow LearntCounter + 1$
    for $v \in conflictSide \cup learntClauseVars$ do
        $Participated_v \leftarrow Participated_v + 1$
    if $\alpha > 0.06$ then
        $\alpha \leftarrow \alpha - 10^{-6}$

procedure ONASSIGN($v \in Vars$)                     ▷ Called when $v$ is assigned by branching or propagation.
    $Assigned_v \leftarrow LearntCounter$
    $Participated_v \leftarrow 0$

procedure ONUNASSIGN($v \in Vars$)                   ▷ Called when $v$ is unassigned by backtracking or restart.
    $Interval \leftarrow LearntCounter - Assigned_v$
    if $Interval > 0$ then                            ▷ $Interval = 0$ is possible due to restarts.
        $r \leftarrow Participated_v / Interval.$     ▷ $r$ is the LR.
        $Q_v = (1 - \alpha) \cdot Q_v + \alpha \cdot r$   ▷ Update the EMA incrementally.

function PICKBRANCHLIT                               ▷ Called when the solver requests the next branching variable.
    $U \leftarrow \{v \in Vars \mid isUnassigned(v)\}$
    return $argmax_{v \in U} Q_v$                     ▷ Use a priority queue for better performance.
```

number of learnt clauses generated in interval I. The *reason side rate* (RSR) of variable v at interval I is defined as $\frac{A(v,I)}{L(I)}$.

Recall that in ERWA, the estimates are updated incrementally as $Q_v \leftarrow (1 - \alpha) \cdot Q_v + \alpha \cdot r$ where r is the LR of v. This extension modifies the update to $Q_v \leftarrow (1 - \alpha) \cdot Q_v + \alpha \cdot (r + \frac{A(v,I)}{L(I)})$ where $\frac{A(v,I)}{L(I)}$ is the RSR of v (see line 20 in Algorithm 2). Note that we did not change the definition of the reward. The extension simply encourages the algorithm to select variables with high RSR when deciding to branch. We hypothesize that variables observed to have high RSR are likely to have high LR as well.

5.3 Extension: Locality

Recent research shows that VSIDS exhibits locality [20], defined with respect to the community structure of the input CNF instance [1,20,25]. Intuitively, if the solver is currently working within a community, it is best to continue focusing on the same community rather than exploring another. We hypothesize that high LR variables also exhibit locality, that is, the branching heuristic can achieve higher LR by restricting exploration.

Algorithm 2. Pseudocode for ERWA as a branching heuristic with the RSR extension. The pseudocode *Algorithm*1.*method*(...) is calling out to the code in Algorithm 1. The procedure *PickBranchLit* is unchanged.

```
 1: procedure Initialize
 2:     $Algorithm1.Initialize()$
 3:     for $v \in Vars$ do                    ▷ $Vars$ is the set of Boolean variables in the input CNF.
 4:         $Reasoned_v \leftarrow 0$          ▷ The number of learnt clauses $v$ reasoned in generating since $Assigned_v$.
 5:
 6: procedure AfterConflictAnalysis($learntClauseVars \subseteq Vars$, $conflictSide \subseteq Vars$)
 7:     $Algorithm1.AfterConflictAnalysis(learntClauseVars, conflictSide)$
 8:     for $v \in (\bigcup_{u \in learntClauseVars} reason(u)) \setminus learntClauseVars$ do
 9:         $Reasoned_v \leftarrow Reasoned_v + 1$
10:
11: procedure OnAssign($v \in Vars$)
12:     $Algorithm1.OnAssign()$
13:     $Reasoned_v \leftarrow 0$
14:
15: procedure OnUnassign($v \in Vars$)
16:     $Interval \leftarrow LearntCounter - Assigned_v$
17:     if $Interval > 0$ then                 ▷ $Interval = 0$ is possible due to restarts.
18:         $r \leftarrow Participated_v / Interval.$          ▷ $r$ is the LR.
19:         $rsr \leftarrow Reasoned_v / Interval.$            ▷ $rsr$ is the RSR.
20:         $Q_v = (1 - \alpha) \cdot Q_v + \alpha \cdot (r + rsr)$   ▷ Update the EMA incrementally.
```

Algorithm 3. Pseudocode for ERWA as a branching heuristic with the locality extension. *AfterConflictAnalysis* is the only procedure modified.

```
1: procedure AfterConflictAnalysis($learntClauseVars \subseteq Vars$, $conflictSide \subseteq Vars$)
2:     $Algorithm2.AfterConflictAnalysis(learntClauseVars, conflictSide)$
3:     $U \leftarrow \{v \in Vars \mid isUnassigned(v)\}$
4:     for $v \in U$ do
5:         $Q_v \leftarrow 0.95 \times Q_v.$
```

Inspired by the VSIDS decay, this extension multiplies the Q_v of every unassigned variable v by 0.95 after each conflict (see line 5 in Algorithm 3). Again, we did not change the definition of the reward. The extension simply discourages the algorithm from exploring inactive variables. This extension is similar to the decay reinforcement model [13,32] where unplayed arms are penalized by a multiplicative decay. The implementation is optimized to do the multiplications in batch. For example, suppose variable v is unassigned for k conflicts. Rather than executing k updates of $Q_v \leftarrow 0.95 \times Q_v$, the implementation simply updates once using $Q_v \leftarrow 0.95^k \times Q_v$.

5.4 Putting it all Together to Obtain the Learning Rate Branching (LRB) Heuristic

The *learning rate branching* (LRB) heuristic refers to ERWA in the MAB abstraction with the RSR and locality extensions. We show that LRB is better at optimizing LR than the other branching heuristics considered, and subsequently has the best overall performance of the bunch.

6 Experimental Results

In this section, we discuss the detailed and comprehensive experiments we performed to evaluate LRB. First, we justify the extensions of LRB by demonstrating their performance vis-a-vis improvements in learning rate. Second, we show that LRB outperforms the state-of-the-art VSIDS and CHB branching heuristic. Third, we show that LRB achieves higher rewards/LR than VSIDS, CHB, and LRB sans the extensions. Fourth, we show the effectiveness of LRB within a state-of-the-art CDCL solver, namely, CryptoMiniSat [29]. To better gauge the results of these experiments, we quote two leading SAT solver developers, Professors Audemard and Simon [3]:

> "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."

6.1 Setup

The experiments are performed by running CDCL solvers with various branching heuristics on StarExec [30], a platform designed for evaluating logic solvers. The StarExec platform uses the Intel Xeon CPU E5-2609 at 2.40 GHz with 10240 KB cache and 24 GB of main memory, running on Red Hat Enterprise Linux Workstation release 6.3, and Linux kernel 2.6.32-431.1.2.el6.x86_64. The benchmarks for the experiments consist of all the instances from the previous 4 SAT Competitions (2014, 2013, 2011, and 2009), in both the application and hard combinatorial categories. For each instance, the solver is given 5000 seconds of CPU time and 7.5 GB of RAM, abiding by the SAT Competition 2013 limits.

Our experiments test different branching heuristics on a *base* CDCL solver, where the only modification is to the branching heuristic to give a fair apple-to-apple comparison. Our base solver is MiniSat version 2.2.0 [12] (simp version) with one modification to use the popular *aggressive LBD-based clause deletion* proposed by the authors of the Glucose solver in 2009 [2]. Since MiniSat is a relatively simple solver with very few features, it is ideal for our base solver to better isolate the effects swapping branching heuristics in our experiments. Additionally, MiniSat is the basis of many competitive solvers and aggressive LBD-based clause deletion is almost universally implemented, hence we believe the results of our experiments will generalize to other solver implementations.

6.2 Experiment: Efficacy of Extensions to ERWA

In this experiment, we demonstrate the effectiveness of the extensions we proposed for LRB. We modified the base solver by replacing the VSIDS branching heuristic with ERWA. We then created two additional solvers, one with the RSR extension and another with both the RSR and locality extensions. We ran these

Table 1. Comparison of our extensions on the base CDCL solver (MiniSat 2.2 with aggressive LBD-based clause deletion). The entries show the number of instances solved for the given solver and benchmark, the higher the better. Green is best, red is worst.

Benchmark	Status	ERWA	ERWA+ RSR	ERWA+ RSR + Locality (LRB)
2009 Application	SAT	85	84	85
	UNSAT	122	120	121
	BOTH	207	204	206
2009 Hard Combinatorial	SAT	98	99	101
	UNSAT	65	68	69
	BOTH	163	167	170
2011 Application	SAT	105	105	103
	UNSAT	98	101	98
	BOTH	203	206	201
2011 Hard Combinatorial	SAT	95	88	93
	UNSAT	45	61	65
	BOTH	140	149	158
2013 Application	SAT	125	133	132
	UNSAT	89	95	95
	BOTH	214	228	227
2013 Hard Combinatorial	SAT	113	110	116
	UNSAT	97	108	110
	BOTH	210	218	226
2014 Application	SAT	111	108	116
	UNSAT	82	77	77
	BOTH	193	185	193
2014 Hard Combinatorial	SAT	87	92	91
	UNSAT	73	87	89
	BOTH	160	179	180
TOTAL (excluding duplicates)	**SAT**	**638**	**632**	**654**
	UNSAT	**574**	**619**	**625**
	BOTH	**1212**	**1251**	**1279**

3 solvers over the entire benchmark and report the number of instances solved by these solvers within the time limit in Table 1. ERWA solves a total of 1212 instances, ERWA with the RSR extension solves a total of 1251 instances, and ERWA with the RSR and locality extensions (i.e., LRB) solves a total of 1279 instances. See Fig. 1 for a cactus plot of the solving times.

6.3 Experiment: LRB vs VSIDS vs CHB

In this experiment, we compare LRB with VSIDS [24] and CHB [19]. Our base solver is MiniSat 2.2 which already implements VSIDS. We then replaced VSIDS in the base solver with LRB and CHB to derive 3 solvers in total, with the only difference being the branching heuristic. We ran these 3 solvers on the entire benchmark and present the results in Table 2. LRB solves a total of 1279 instances, VSIDS solves a total of 1179 instances, and CHB solves a total of 1235 instances. See Fig. 1 for a cactus plot of the solving times.

Table 2. Apple-to-apple comparison between branching heuristics (LRB, CHB, and VSIDS) in a version of MiniSat 2.2 with aggressive LBD-based clause deletion. The entries show the number of instances in the benchmark the given branching heuristic solves, the higher the better. Green is best, red is worst. The LRB version (we dub as MapleSAT), outperforms the others.

Benchmark	Status	LRB	VSIDS	CHB
2009 Application	SAT	85	83	89
	UNSAT	121	125	119
	BOTH	206	208	208
2009 Hard Combinatorial	SAT	101	100	103
	UNSAT	69	66	67
	BOTH	170	166	170
2011 Application	SAT	103	95	106
	UNSAT	98	99	96
	BOTH	201	194	202
2011 Hard Combinatorial	SAT	93	88	102
	UNSAT	65	48	47
	BOTH	158	136	149
2013 Application	SAT	132	127	137
	UNSAT	95	86	79
	BOTH	227	213	216
2013 Hard Combinatorial	SAT	116	115	122
	UNSAT	110	73	96
	BOTH	226	188	218
2014 Application	SAT	116	105	115
	UNSAT	77	94	73
	BOTH	193	199	188
2014 Hard Combinatorial	SAT	91	91	90
	UNSAT	89	59	76
	BOTH	180	150	166
TOTAL (excluding duplicates)	**SAT**	**654**	**626**	**673**
	UNSAT	**625**	**553**	**562**
	BOTH	**1279**	**1179**	**1235**

6.4 Experiment: LRB and Learning Rate

In this experiment, we measure the efficacy of the 5 branching heuristics from Tables 1 and 2 at maximizing the LR. For each instance in the benchmark, we solve the instance 5 times with the 5 branching heuristics implemented in the base solver. For each branching heuristic, we track all the observed rewards (i.e., LR) and record the mean observed reward at the end of the run, regardless if the solver solves the instance or not. We then rank the 5 branching heuristics by their mean observed reward for that instance. A branching heuristic gets a rank of 1 (resp. 5) if it has the highest (resp. lowest) mean observed reward for that instance. For each branching heuristic, we then average its ranks over the entire benchmark and report these numbers in Table 3. The experiment shows that LRB is the best heuristic in terms of maximizing the reward LR (corresponding to a rank closest to 1) in almost every category. In addition, the experiment shows that the RSR and locality extensions increase the observed rewards relative to vanilla ERWA. Somewhat surprisingly, VSIDS and CHB on average observe higher rewards (i.e., LR) than ERWA, despite the fact that VSIDS and CHB are designed without LR as an explicit objective. This perhaps partly explains the effectiveness of those two heuristics.

Table 3. The average ranking of observed rewards compared between different branching heuristics in MiniSat 2.2 with aggressive LBD-based clause deletion. The lower the reported number, the better the heuristic is at maximizing the observed reward relative to the others. Green is best, red is worst.

Benchmark	Status	LRB	ERWA	ERWA + RSR	VSIDS	CHB
2009 Application	SAT	2.41	3.79	3.42	2.51	2.87
	UNSAT	2.13	4.16	3.32	2.90	2.49
	BOTH	2.25	4.01	3.36	2.74	2.65
2009 Hard Combinatorial	SAT	2.43	3.30	3.03	3.29	2.95
	UNSAT	2.18	4.18	3.48	3.22	1.94
	BOTH	2.33	3.66	3.21	3.26	2.53
2011 Application	SAT	2.25	3.61	3.02	2.77	3.35
	UNSAT	2.14	3.82	3.22	3.49	2.33
	BOTH	2.20	3.72	3.12	3.13	2.85
2011 Hard Combinatorial	SAT	2.57	3.47	2.98	3.46	2.53
	UNSAT	2.57	3.72	3.32	3.54	1.85
	BOTH	2.57	3.56	3.11	3.49	2.27
2013 Application	SAT	2.33	3.60	3.16	2.49	3.41
	UNSAT	2.02	4.16	3.07	3.39	2.37
	BOTH	2.19	3.85	3.12	2.89	2.95
2013 Hard Combinatorial	SAT	2.51	3.57	2.91	3.03	2.98
	UNSAT	1.99	3.92	2.65	4.26	2.18
	BOTH	2.24	3.75	2.78	3.65	2.58
2014 Application	SAT	2.27	3.68	3.21	2.50	3.35
	UNSAT	2.24	4.34	3.20	2.82	2.40
	BOTH	2.25	4.01	3.21	2.66	2.88
2014 Hard Combinatorial	SAT	2.43	3.51	3.03	2.78	3.26
	UNSAT	1.81	4.38	2.69	3.82	2.30
	BOTH	2.11	3.96	2.85	3.31	2.76
TOTAL (excluding duplicates)	SAT	**2.45**	**3.53**	**3.10**	**2.72**	**3.20**
	UNSAT	**2.12**	**4.08**	**3.10**	**3.41**	**2.30**
	BOTH	**2.28**	**3.81**	**3.10**	**3.07**	**2.74**

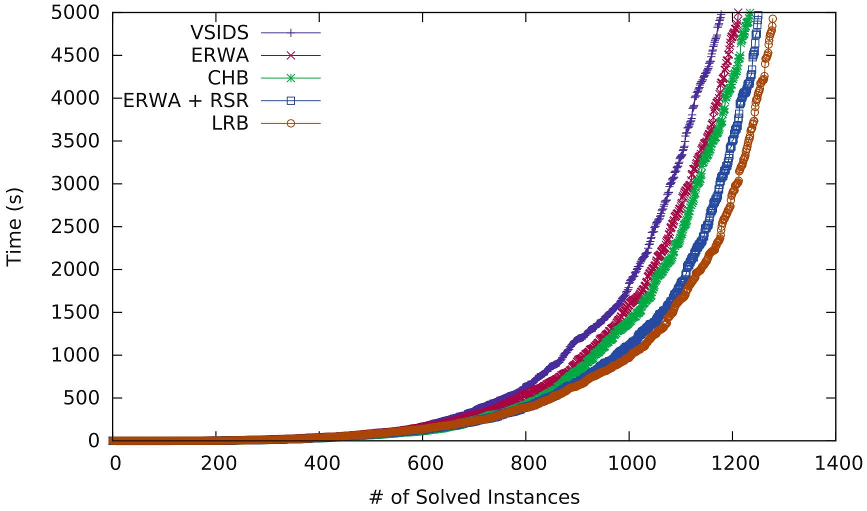

Fig. 1. A cactus plot of the 5 branching heuristics in MiniSat 2.2 with aggressive LBD-based clause deletion. The benchmark consists of the 4 most recent SAT Competition benchmarks (2014, 2013, 2011, 2009) including both the application and hard combinatorial categories, excluding duplicate instances. A point (x, y) on the plot is interpretted as: y instances in the benchmark took less than x seconds to solve for the branching heuristic. The further right and further down, the better.

6.5 Experiment: LRB vs State-of-the-Art CDCL

In this experiment, we test how LRB-enchanced CryptoMiniSat competes against the state-of-the-art solvers CryptoMiniSat [29], Glucose [4], and Lingeling [6] which all implement VSIDS. We modified CryptoMiniSat 4.5.3 [29] by replacing VSIDS with LRB, leaving everything else unmodified. We ran unmodified CryptoMiniSat, Glucose, and Lingeling, along with the LRB-enchanced CryptoMiniSat on the benchmark and report the results in Table 4. LRB improved CryptoMiniSat on 6 of the 8 benchmarks and solves 59 more instances overall.

7 Related Work

The Chaff solver introduced the VSIDS branching heuristic in 2001 [24]. Although many branching heuristics have been proposed [7,15–17,22,28], VSIDS and its variants remain as the dominant branching heuristic employed in modern CDCL SAT solvers. Carvalho and Marques-Silva used rewards based on learnt clause length and backjump size to improve VSIDS [10]. More precisely, the bump value of VSIDS is increased for short learnt clauses and/or long backjumps. Their usage of rewards is unrelated to the definition of rewards in the reinforcement learning and multi-armed bandits context. Lagoudakis and Littman used reinforcement learning to dynamically switch between a fixed set of 7 well-known

Table 4. Apple-to-apple comparison between four state-of-art solvers: CryptoMiniSat (CMS) with LRB heuristic, CMS with VSIDS, Glucose, and Lingeling. The table shows the number of instances solved per SAT Competition benchmark, categorized as SAT or UNSAT instances. CMS with LRB (we dub as MapleCMS) outperforms CMS with VSIDS on most benchmarks.

Benchmark	Status	CMS with LRB	CMS with VSIDS	Glucose	Lingeling
2009 Application	SAT	85	87	83	80
	UNSAT	140	143	138	141
	BOTH	225	230	221	221
2009 Hard Combinatorial	SAT	102	95	90	98
	UNSAT	71	65	70	83
	BOTH	173	160	160	181
2011 Application	SAT	106	97	94	94
	UNSAT	122	129	127	134
	BOTH	228	226	221	228
2011 Hard Combinatorial	SAT	86	86	80	88
	UNSAT	57	49	44	66
	BOTH	143	135	124	154
2013 Application	SAT	115	109	104	100
	UNSAT	120	115	111	122
	BOTH	235	224	215	222
2013 Hard Combinatorial	SAT	116	114	115	114
	UNSAT	114	101	106	117
	BOTH	230	215	221	231
2014 Application	SAT	107	102	99	101
	UNSAT	118	127	120	141
	BOTH	225	229	219	242
2014 Hard Combinatorial	SAT	89	85	79	89
	UNSAT	122	100	93	119
	BOTH	211	185	172	208
TOTAL (excluding duplicates)	**SAT**	**619**	**598**	**575**	**589**
	UNSAT	**738**	**700**	**685**	**782**
	BOTH	**1357**	**1298**	**1260**	**1371**

SAT branching heuristics [18]. Their technique requires offline training on a class of similar instances. Loth et al. used multi-armed bandits for directing the growth of the search tree for Monte-Carlo Tree Search [21]. The rewards are computed based on the relative depth failure of the tree walk. Fröhlich et al. used the UCB algorithm from multi-armed bandits to select the candidate variables to define the neighborhood of a stochastic local search for the theory of bitvectors [14]. The rewards they are optimizing is to minimize the number of unsatisfied clauses. Liang et al. also applied ERWA as a branching heuristic called CHB [19]. As stated earlier, CHB is neither an optimization nor learning algorithm since the rewards are computed on past events.

8 Conclusions and Future Work

In this paper, we provide three main contributions, and each has potential for further enhancements.

Contribution I: We define LR as a metric for the branching heuristic to optimize. LR captures the intuition that the branching heuristic should assign variables which are likely to generate a high quantity of learnt clauses with no regards to the "quality" of those clauses [2]. A new metric that captures quality should encourage better clause learning. Or perhaps branching heuristics can be stated as a multi-objective optimization problem where a good heuristic would balance the tradeoff between quality and quantity of learnt clauses.

Additionally, we would like to stress that the starting point for this research was a model of CDCL SAT solvers as an interplay between branching heuristic and clause learning. The branching heuristic guides the search, and has great impact on the clauses that will be learnt during the run of the solver. In the reverse direction, clause learning provides feedback to guide branching heuristics like VSIDS, CHB, and LRB. We plan to explore a mathematical model where the branching heuristic is an inductive engine (machine learning), and the conflict analysis is a deductive feedback mechanism. Such a model could enable us to prove complexity theoretic results that at long last might explain why CDCL SAT solvers are so efficient for industrial instances.

Contribution II: We chose MAB as the optimization method in this paper, but many other optimization techniques can be applied to optimize LR. The most natural extension to our work here is to incorporate the internal state of the solver and apply stateful reinforcement learning. The state, for example, could be the current community the solver is focused on and exploiting this information could improve the locality of the branching heuristic [20].

Contribution III: We based LRB on one MAB algorithm, ERWA, due to its low computational overhead. The literature of multi-armed bandits is very rich, and provides many alternative algorithms with a wide spectrum of characteristics and assumptions. It is fruitful to explore the MAB literature to determine the best algorithm for branching in CDCL SAT solvers.

Finally, as our experimental results suggest, the line of research we have just started exploring, namely, branching heuristics as machine learning algorithms (and branching as an optimization problem) has already shown considerable improvement over previous state-of-the-art branching heuristics such as VSIDS and CHB, and affords a rich design space of heuristics to explore in the future.

References

1. Ansótegui, C., Giráldez-Cru, J., Levy, J.: The community structure of SAT formulas. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 410–423. Springer, Heidelberg (2012)
2. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Proceedings of the 21st International Jont Conference on Artifical Intelligence, IJCAI 2009, pp. 399–404. Morgan Kaufmann Publishers Inc., San Francisco (2009)
3. Audemard, G., Simon, L.: Refining restarts strategies for SAT and UNSAT. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 118–126. Springer, Heidelberg (2012)

4. Audemard, G., Simon, L.: Glucose 2.3 in the SAT 2013 Competition. In: Proceedings of SAT Competition 2013, pp. 42–43 (2013)
5. Biere, A.: Adaptive restart strategies for conflict driven SAT solvers. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 28–33. Springer, Heidelberg (2008)
6. Biere, A.: Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. FMV Report Series Technical report 10(1) (2010)
7. Biere, A., Fröhlich, A.: Evaluating CDCL variable scoring schemes. In: Heule, M., Weaver, S. (eds.) SAT 2015. LNCS, vol. 9340, pp. 405–422. Springer, Heidelberg (2015). doi:10.1007/978-3-319-24318-4_29
8. Brown, R.G.: Exponential smoothing for predicting demand. Oper. Res. **5**, 145–145 (1957)
9. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, pp. 322–335. ACM, New York (2006)
10. Carvalho, E., Marques-Silva, J.P.: Using rewarding mechanisms for improving branching heuristics. In: Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (2004)
11. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Form. Methods Syst. Des. **19**(1), 7–34 (2001)
12. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
13. Erev, I., Roth, A.E.: Predicting how people play games: reinforcement learning in experimental games with unique, mixed strategy equilibria. Am. Econ. Rev. **88**(4), 848–881 (1998)
14. Fröhlich, A., Biere, A., Wintersteiger, C., Hamadi, Y.: Stochastic local search for satisfiability modulo theories. In: Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, AAAI 2015, pp. 1136–1143. AAAI Press (2015)
15. Gershman, R., Strichman, O.: HaifaSat: a new robust SAT solver. In: Ur, S., Bin, E., Wolfsthal, Y. (eds.) HVC 2005. LNCS, vol. 3875, pp. 76–89. Springer, Heidelberg (2006)
16. Goldberg, E., Novikov, Y.: BerkMin: a fast and robust sat-solver. Discrete Appl. Math. **155**(12), 1549–1561 (2007)
17. Jeroslow, R.G., Wang, J.: Solving propositional satisfiability problems. Ann. Math. Artif. Intell. **1**(1–4), 167–187 (1990)
18. Lagoudakis, M.G., Littman, M.L.: Learning to select branching rules in the DPLL procedure for satisfiability. Electron. Notes Discrete Math. **9**, 344–359 (2001)
19. Liang, J.H., Ganesh, V., Poupart, P., Czarnecki, K.: Exponential recency weighted average branching heuristic for SAT solvers. In: Proceedings of AAAI 2016 (2016)
20. Liang, J.H., Ganesh, V., Zulkoski, E., Zaman, A., Czarnecki, K.: Understanding VSIDS branching heuristics in conflict-driven clause-learning SAT solvers. In: Liang, J.H., Ganesh, V., Zulkoski, E., Zaman, A., Czarnecki, K. (eds.) HVC 2015. LNCS, vol. 9434, pp. 225–241. Springer, Heidelberg (2015). doi:10.1007/978-3-319-26287-1_14
21. Loth, M., Sebag, M., Hamadi, Y., Schoenauer, M.: Bandit-based search for constraint programming. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 464–480. Springer, Heidelberg (2013)
22. Marques-Silva, J.: The impact of branching heuristics in propositional satisfiability algorithms. In: Barahona, P., Alferes, J.J. (eds.) EPIA 1999. LNCS (LNAI), vol. 1695, pp. 62–74. Springer, Heidelberg (1999)

23. Marques-Silva, J.P., Sakallah, K.A.: GRASP-a new search algorithm for satisfiability. In: Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design, ICCAD 1996, pp. 220–227. IEEE Computer Society, Washington, DC (1996)
24. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proceedings of the 38th Annual Design Automation Conference, DAC 2001, pp. 530–535. ACM, New York (2001)
25. Newsham, Z., Ganesh, V., Fischmeister, S., Audemard, G., Simon, L.: Impact of community structure on SAT solver performance. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 252–268. Springer, Heidelberg (2014)
26. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 294–299. Springer, Heidelberg (2007)
27. Rintanen, J.: Planning and SAT. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, vol. 185, pp. 483–504. IOS Press, Amsterdam (2009)
28. Ryan, L.: Efficient Algorithms for Clause-Learning SAT Solvers. Master's thesis, Simon Fraser University (2004)
29. Soos, M.: CryptoMiniSat v4. In: SAT Competition, p. 23 (2014)
30. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: a cross-community infrastructure for logic solving. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS, vol. 8562, pp. 367–373. Springer, Heidelberg (2014)
31. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction, vol. 1. MIT Press Cambridge, Massachusetts (1998)
32. Yechiam, E., Busemeyer, J.R.: Comparison of basic assumptions embedded in learning models for experience-based decision making. Psychon. Bull. Rev. **12**(3), 387–402 (2005)
33. Zhang, L., Madigan, C.F., Moskewicz, M.H., Malik, S.: Efficient conflict driven learning in a boolean satisfiability solver. In: Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design, ICCAD 2001, pp. 279–285. IEEE Press, Piscataway (2001)