An Empirical Study of Branching Heuristics through the Lens of Global Learning Rate

Jia Hui Liang[1], Hari Govind V K[2],
Pascal Poupart[1], Krzysztof Czarnecki[1], and Vijay Ganesh[1]

[1] University of Waterloo, Waterloo, Canada
[2] College Of Engineering, Thiruvananthapuram, India

Abstract. In this paper, we analyze a suite of 7 well-known branching heuristics proposed by the SAT community and show that the better heuristics tend to generate more learnt clauses per decision, a metric we define as the global learning rate (GLR). Like our previous work on the LRB branching heuristic, we once again view these heuristics as techniques to solve the *learning rate* optimization problem. First, we show that there is a strong positive correlation between GLR and solver efficiency for a variety of branching heuristics. Second, we test our hypothesis further by developing a new branching heuristic that *maximizes GLR* greedily. We show empirically that this heuristic achieves very high GLR and interestingly very low literal block distance (LBD) over the learnt clauses. In our experiments this greedy branching heuristic enables the solver to solve instances faster than VSIDS, when the branching time is taken out of the equation. This experiment is a good proof of concept that a branching heuristic maximizing GLR will lead to good solver performance modulo the computational overhead. Third, we propose that machine learning algorithms are a good way to cheaply approximate the greedy GLR maximization heuristic as already witnessed by LRB. In addition, we design a new branching heuristic, called SGDB, that uses a stochastic gradient descent online learning method to dynamically order branching variables in order to maximize GLR. We show experimentally that SGDB performs on par with the VSIDS branching heuristic.

1 Introduction

Searching through a large, potentially exponential, search space is a reoccurring problem in many fields of computer science. Rather than reinventing the wheel and implementing complicated search algorithms from scratch, many researchers in fields as diverse as software engineering [7], hardware verification [9], and AI [16] have come to rely on SAT solvers as a general purpose tool to efficiently search through large spaces. By reducing the problem of interest down to a Boolean formula, engineers and scientists can leverage off-the-shelf SAT solvers to solve their problems without needing expertise in SAT or developing special-purpose algorithms. Modern conflict-driven clause-learning (CDCL) SAT solvers can solve a wide-range of practical problems with surprising efficiency, thanks to decades of ongoing research by the SAT community. Two notable milestones that are key to the success of SAT solvers are the Variable State Independent Decaying Sum (VSIDS) branching heuristic (and its variants) [23] and conflict analysis techniques [22]. The VSIDS branching heuristic has been the dominant branching heuristic since 2001, evidenced by its presence in most competitive solvers such as Glucose [4], Lingeling [5], and CryptoMiniSat [26].

One of the challenges in designing branching heuristics is that it is not clear what constitutes a good decision variable. We proposed one solution to this issue in our LRB branching heuristic paper [19], which is to frame branching as an optimization problem. We defined a computable metric called *learning rate* and defined the objective as maximizing the learning rate. Good decision variables are ones with high learning rate. Since learning rate is expensive to compute a priori, we used a multi-armed bandit learning algorithm to estimate the learning rate on-the-fly as the basis for the LRB branching heuristic [19].

In this paper, we deepen our previous work and our starting point remains the same, namely, branching heuristics should be designed to solve the optimization problem of maximizing learning rate. In LRB, the learning rate metric is defined per variable. In this paper, we define a new metric, called the *global learning rate* (GLR) to measure the solver's overall propensity to generate conflicts, rather than the variable-specific metric we defined in the case of LRB. Our experiments demonstrate that GLR is an excellent objective to maximize.

1.1 Contributions

1. **A new objective for branching heuristic optimization:** In our previous work with LRB, we defined a metric that measures learning rate per variable. In this paper, we define a metric called the global learning rate (GLR), that measures the number of learnt clauses generated by the solver per decision, which intuitively is a better metric to optimize since it measures the solver as a whole. We show that the objective of maximizing GLR is consistent with our knowledge of existing branching heuristics, that is, the faster branching heuristics tend to achieve higher GLR. We perform extensive experiments over 7 well-known branching heuristics to establish the correlation between high GLR and better solver performance. (Section 3)
2. **A new branching heuristic to greedily maximize GLR:** To further scientifically test the conjecture that GLR maximization is a good objective, we design a new branching heuristic that greedily maximizes GLR by always selecting decision variables that cause immediate conflicts. It is greedy in the sense that it optimizes for causing immediate conflicts, and it does not consider future conflicts as part of its scope. Although the computational overhead of this heuristic is very high, the variables it selects are "better" than VSIDS. More precisely, if we ignore the computation time to compute the branching variables, the greedy branching heuristic generally solves more instances faster than VSIDS. Another positive side-effect of the greedy branching heuristic is that relative to VSIDS, it has lower learnt clause literal block distance (LBD) [3], a sign that it is learning higher quality clauses. The

combination of learning faster (due to higher GLR) and learning better (due to lower LBD) clauses explains the power of the greedy branching heuristic. Globally optimizing the GLR considering all possible future scenarios a solver can take is simply too prohibitive. Hence, we limited our experiments to the greedy approach. Although this greedy branching heuristic takes too long to select variables in practice, it gives us a gold standard of what we should aim for. We try to approximate it as closely as possible in our third contribution. (Section 4)

3. **A new machine learning branching heuristic to maximize GLR:** We design a second heuristic, called stochastic gradient descent branching (SGDB), using machine learning to approximate our gold standard, the greedy branching heuristic. SGDB trains an online logistic regression model by observing the conflict analysis procedure as the CDCL algorithm solves an instance. As conflicts are generated, SGDB will update the model to better fit its observations. Concurrently, SGDB also uses this model to rank variables based on their likelihood to generate conflicts if branched on. We show that in practice, SGDB is on par with the VSIDS branching heuristic over a large and diverse benchmark but still shy of LRB. However, more work is required to improve the learning in SGDB. (Section 5)

2 Background

Clause Learning: Clause learning produces a new clause after each conflict to prevent the same or similar conflicts from reoccurring [22]. This requires maintaining an implication graph where the nodes are assigned literals and edges are implications forced by Boolean constraint propagation (BCP). When a clause is falsified, the CDCL solver invokes *conflict analysis* to produce a *learnt clause* from the conflict. It does so by *cutting* the implication graph, typically at the first-UIP [22], into the *reason side* and the *conflict side* with the condition that the decision variables appear on the reason side and the falsified clause appears on the conflict side. A new learnt clause is constructed by negating the reason side literals incident to the cut. Literal block distance (LBD) is a popular metric for measuring the "quality" of a learnt clause [3]. The lower the LBD the better.
Supervised Learning: Suppose there exists some function $f : Input \rightarrow Output$ that we do not have the code for. However, we do have *labeled* training data in the form of $\langle Input_i, f(Input_i)\rangle$ pairs. Given a large set of these labeled training data, also called a training set, there exists machine learning algorithms that can infer a new function $\tilde{f}$ that approximates f. These types of machine learning algorithms are called *supervised learning* algorithms. If everything goes well, $\tilde{f}$ will return the correct output with a high probability when given inputs from the training set, in which case we we say $\tilde{f}$ fits the training set. Ideally, $\tilde{f}$ will also return the correct output for inputs that are not in the training set, in which case we say the function generalizes.

Most supervised learning algorithms require the input data to be represented as a vector of numbers. Feature extraction solves this issue by transforming each input data into a vector of real numbers, called a feature vector, that summarizes the input datum. During training, the feature vectors are used for training in place of the original input, hence learning the function $\tilde{f} : \mathbb{R}^n \rightarrow Output$ where $\mathbb{R}^n$ is the feature vector's type. Deciding which features to extract has a large impact on the learning algorithm's success.

In this paper, we only consider a special subclass of supervised learning called binary classification. In other words, the function we want to learn has the type $f : Input \rightarrow \{1, 0\}$, hence f maps every input to either the class 1 or the class 0.

We use logistic regression [10], a popular technique for binary classification, to learn a function $\tilde{f}$ that cheaply approximates f. The function learned by logistic regression has the type $\tilde{f} : \mathbb{R}^n \rightarrow [0, 1]$ where $\mathbb{R}^n$ is from the feature extraction and the output is a probability in $[0, 1]$ that the input is in class 1. Logistic regression defines the function $\tilde{f}$ as follows.

$$\tilde{f}([x_1, x_2, ..., x_n]) := \sigma(w_0 + w_1 x_1 + w_2 x_2 + ... + w_n x_n), \quad \sigma(z) := \frac{1}{1 + e^{-z}}$$

The weights $w_i \in \mathbb{R}$ measure the significance of each feature. The learning algorithm is responsible for finding values for these weights to make $\tilde{f}$ approximate f as closely as possible. The sigmoid function σ simply squeezes the linear function to be between 0 and 1. Hence $\tilde{f}$ outputs a real number between 0 and 1, which is expected since it is a probability.

The learning algorithm we use to set the weights is called stochastic gradient descent (SGD) [6], which is a popular algorithm for logistic regression. SGD minimizes the misclassification rate by taking a step in the opposite direction of the gradient with respect to each data point. The misclassification rate of a data point can be computed by the following error function:

$$Err(\mathbf{x}, y, \mathbf{W}) = y(1 - \tilde{f}(\mathbf{x}; \mathbf{W})) + (1 - y)(\tilde{f}(\mathbf{x}; \mathbf{W}))$$

where $\mathbf{x}$ is the input of a data point, y is the corresponding target class (0 or 1) for this data point and $\mathbf{W}$ is a vector weights. SGD takes a step in the opposite direction of the gradient as follows:

$$\mathbf{W}' \leftarrow \mathbf{W} - \alpha \frac{\partial Err(\mathbf{x}, y, \mathbf{W})}{\partial \mathbf{W}}$$

Here α is the step length (also known as the learning rate, not to be confused with the unrelated definition of learning rate in LRB). Under normal conditions, $\tilde{f}$ with the new weights $\mathbf{W}'$ will fit the training set better than with the old weights $\mathbf{W}$. If training time is not an issue, then SGD can be applied repeatedly until a fixed point is reached. The parameter $0 < \alpha < 1$ controls how aggressively the technique converges.

A common problem with machine learning in general is overfitting, where the trained function $\tilde{f}$ predicts correctly for the inputs it has seen in the training set, but works poorly for inputs it has not seen. We use a common technique called L2 regularization [24] to mitigate overfitting. L2 regularization introduces a new term in the error function that favors small weights

$$Err(\mathbf{x}, y, \mathbf{W}) = y(1 - \tilde{f}(\mathbf{x}; \mathbf{W})) + (1 - y)(\tilde{f}(\mathbf{x}; \mathbf{W})) + \lambda ||\mathbf{W}||_2^2$$

Here λ is a parameter that determines the importance of the regularization penalty. How this prevents overfitting is beyond the scope of this paper.

SGD is also commonly used in an online fashion. Each time new data comes in, SGD is applied to this new data to update the weights, then the data is discarded. This has two advantages. Discarding the data keeps the memory usage low, especially useful when data is abundant. Additionally, the distribution in which the data is created can change over time. Online stochastic gradient does not assume the distribution is fixed and adjusts the weights accordingly after enough time. These two advantages are critical in our use of SGD.

3 GLR Maximization as a Branching Heuristic Objective

We framed the branching heuristic as an optimization problem in our earlier work [19], and we will continue to do so here. Formalizing the problem as an optimization problem opens up the problem to a wide range of existing optimization algorithms, and we exploited this very idea to develop the LRB [19] branching heuristic. The big difference between our previous papers and the current one is that the objective function for optimization in our previous work was learning per variable, whereas here we define it as the global learning rate (GLR) discussed below.

The first step to solving an optimization problem is to define the objective. Ideally the objective of the branching heuristic is to minimize the total running time. However, it is infeasible to calculate the running time a priori, which makes it unsuitable as an objective for branching. Instead, we target an easy to compute feature that correlates with solving time.

We define the *global learning rate* (GLR) of a solver as $GLR := \frac{\#\ of\ conflicts}{\#\ of\ decisions}$. Our goal is to construct a new branching heuristic to maximize the GLR. We assume that one clause is learnt per conflict. Learning multiple clauses per conflict has diminishing returns since they block the same conflict. But before we present our branching heuristic, let us justify why maximizing GLR is a reasonable objective for a branching heuristic. Past research concludes that clause learning is the most important feature for good performance in a CDCL solver [15], so perhaps it is not surprising that increasing the rate at which clauses are learnt is a reasonable objective. In our experiments, we assume the learning scheme is first-UIP since it is universally used by all modern CDCL solvers.

3.1 GLR vs Solving Time

We propose the following hypothesis: *for a given instance, the branching heuristic that achieves higher GLR tends to solve that instance faster than heuristics with lower GLR.* We provide empirical evidence in support of the hypothesis.

In the following experiment, we tested the above hypothesis on 7 branching heuristics: LRB [19], CHB [18], VSIDS (MiniSat [11] variation of VSIDS), CVSIDS (Chaff [23] variation of VSIDS), Berkmin [13], DLIS [21], and Jeroslow-Wang [14]. We created 7 versions of MapleSAT [1], one for each branching heuristic, keeping the code unrelated to the branching heuristic untouched. We ran all

Table 1. The GLR, number of instances solved, and average solving time for 7 different branching heuristics, sorted by the number of solved instances. Timed out runs have a solving time of 1800s in the average solving time.

Heuristic	*Avg LBD*	*Avg GLR*	*# Instances Solved*	*Avg Solving Time(s)*
LRB	10.797	0.533	1552	905.060
CHB	11.539	0.473	1499	924.065
VSIDS	17.163	0.484	1436	971.425
CVSIDS	19.709	0.406	1309	1043.971
BERKMIN	27.485	0.382	629	1446.337
DLIS	20.955	0.318	318	1631.236
JW	176.913	0.173	290	1623.226

7 branching heuristics on each application and hard combinatorial instance from every SAT Competition and SAT Race held between 2009 and 2016 with duplicate instances removed. At the end of each run, we recorded the solving time, GLR at termination, and the average LBD of clauses learnt. All experiments in this paper were conducted on StarExec [28], a platform purposefully designed for evaluating SAT solvers. For each instance, the solver was given 1800 seconds of CPU time and 8GB of RAM. The code for our experiments can be found on the MapleSAT website [2].

The results are presented in Table 1. Note that sorting by GLR in decreasing order, sorting by instances solved in decreasing order, sorting by LBD in increasing order, and sorting by average solving time in increasing order produces essentially the same ranking. This gives credence to our hypothesis that GLR correlates with branching heuristic effectiveness. Additionally, the experiment shows that high GLR correlates with low LBD.

To better understand the correlation between GLR and solving time, we ran a second experiment where for each instance, we computed the Spearman's rank correlation coefficient [27] (Spearman correlation for short) between the 7 branching heuristics' GLR and solving time. We then averaged all the instances' Spearman correlations by applying the Fisher transformation [12] to these correlations, then computing the mean, then applying the inverse Fisher transformation. This is a standard technique in statistics to average over correlations. This second experiment was performed on all the application and hard combinatorial instances from SAT Competition 2013 using the StarExec platform with a 5400s timeout and 8GB of RAM. For this benchmark, the average Spearman correlation is -0.3708, implying a negative correlation between GLR and solving time, or in other words, a high (resp. low) GLR tends to have low (resp. high) solving time as we hypothesized. Table 2 shows the results of the same correlation experiment with different solver configurations. The results show that the correlations remain moderately negative for all the configurations we tried.

Maximizing GLR also makes intuitive sense when viewing the CDCL solver as a proof system. Every conflict generates a new lemma in the proof. Every decision is like a new "case" in the proof. Intuitively, the solver wants to generate lemmas

Table 2. The Spearman correlation relating GLR to solving time between the 7 heuristics. The experiment is repeated with different solver configurations. MapleSAT is the default configuration which is essentially MiniSat [11] with phase saving [25], Luby restarts [20], and rapid clause deletion [3] based on LBD [3]. Clause activity based deletion is the scheme implemented in vanilla MiniSat.

Configuration	*Spearman Correlation*
MapleSAT	-0.3708
No phase saving	-0.4492
No restarting	-0.3636
Clause deletion based on clause activity	-0.4235
Clause deletion based on LBD	-0.3958
Rapid clause deletion based on clause activity	-0.3881

quickly using as few cases as possible, or in other words, maximize conflicts with as few decisions as possible. This is equivalent to maximizing GLR. Of course in practice, not all lemmas/learnt clauses are of equal quality, so the quality is also an important objective. We will comment more on this in later sections.

4 Greedy Maximization of GLR

Finding the globally optimal branching sequence that maximizes GLR is intractable in general. Hence we tackle a simpler problem to maximize GLR greedily instead. Although this is too computationally expensive to be effective in practice, it provides a proof of concept for GLR maximization and a gold standard for subsequent branching heuristics.

We define the function $c : PA \to \{1, 0\}$ that maps partial assignments to either class 1 or class 0. Class 1 is the "conflict class" which means that applying BCP to the input partial assignment with the current clause database would encounter a conflict once BCP hits a fixed-point. Otherwise the input partial assignment is given the class 0 for "non-conflict class". Note that c is a mathematical function with no side-effects, that is applying it does not alter the state of the solver. The function c is clearly decidable via one call to BCP, although it is quite costly when called too often.

The greedy GLR branching (GGB) heuristic is a branching heuristic that maximizes GLR greedily. When it comes time to branch, the branching heuristic is responsible for appending a decision variable (plus a sign) to the current partial assignment. GGB prioritizes decision variables where the new partial assignment falls in class 1 according to the function c. In other words, GGB branches on decision variables that cause a conflict during the subsequent call to BCP, if such variables exist. See Algorithm 1 for the implementation of GGB.

Unfortunately, GGB is very computationally expensive due to the numerous calls to the c function every time a new decision variable is needed. However, we show that GGB significantly increases the GLR relative to the base branching heuristic VSIDS. Additionally, we show that if the time to compute the decision variables was ignored, then GGB would be a more efficient heuristic than

Algorithm 1 Pseudocode for the *GGB* heuristic using the function c to greedily maximize GLR. Note that GGB is a meta-heuristic, it takes an existing branching heuristic (VSIDS in the following pseudocode) and makes it greedier by causing conflicts whenever possible. In general, VSIDS can be replaced with any other branching heuristic.

```
function PhaseSaving(Var)                    ▷ Return the variable plus a sign.
    return $mkLit(Var, Var_{savedPolarity})$

function VSIDS(Vars)        ▷ Return the variable with highest VSIDS activity plus a sign.
    return $PhaseSaving(argmax_{v \in Vars} v_{activity})$

function GGB
    $CPA \leftarrow CurrentPartialAssignment$
    $V \leftarrow UnassignedVariables$
    $oneClass \leftarrow \{v \in V \quad | \quad c(CPA \cup \{PhaseSaving(v)\}) = 1\}$
    $zeroClass \leftarrow V \setminus oneClass$
    if $oneClass \neq \emptyset$ then                 ▷ Next BCP will cause a conflict.
        return VSIDS(oneClass)
    else                                       ▷ Next BCP will not cause a conflict.
        return VSIDS(zeroClass)
```

VSIDS. This suggests we need to cheaply approximate GGB to avoid the heavy computation. A cheap and accurate approximation of GGB would in theory be a better branching heuristic than VSIDS.

4.1 Experimental Results

In this section, we show that GGB accomplishes its goal of increasing the GLR and solving instances faster. Experiments were performed with MapleSAT using the StarExec platform with restarts and clause deletion turned off to minimize the effects of external heuristics. For each of the 300 instances in the SAT Competition 2016 application category, MapleSAT was ran twice, the first run configured with VSIDS and the second run configured with GGB. The run with VSIDS used a timeout of 5000 seconds. The run with GGB used a timeout of 24 hours to account for the heavy computational overhead. We define *effective time* as the solving time minus the time spent by the branching heuristic selecting variables. Figure 1 shows the results of effective time between the two heuristics. Only *comparable* instances are plotted. An instance is comparable if either both heuristics solved the instance or one heuristic solved the instance with an effective time of x seconds while the other heuristic timed out with an effective time greater than x seconds.

Of the comparable instances, GGB solved 69 instances with a lower effective time than VSIDS and 29 instances with a higher effective time. Hence if the branching was free, then GGB would solve instances faster than VSIDS 70% of the time. GGB achieves a higher GLR than VSIDS for all but 2 instances, hence it does a good job increasing GLR as expected. Figure 2 shows the same experiment except the points are colored by the average LBD of all clauses learnt from start until termination. GGB has a lower LBD than VSIDS for 72 of the 98 comparable instances. We believe this is because GGB by design causes conflicts

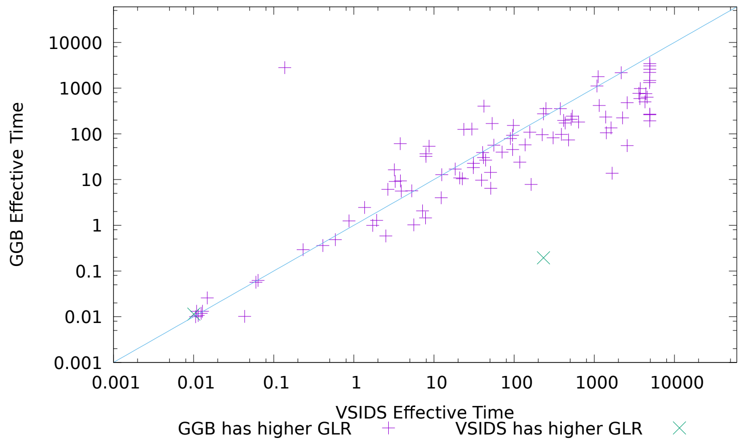

Fig. 1. GGB vs VSIDS. Each point in the plot is a comparable instance. Note that the axes are in log scale. GGB has a higher GLR for all but 2 instances. GGB has a mean GLR of 0.74 for this benchmark whereas VSIDS has a mean GLR of 0.59.

earlier when the decision level is low, which keeps the LBD small since LBD cannot exceed the current decision level.

5 Stochastic Gradient Descent Branching Heuristic

GGB is too expensive in practice due to the computational cost of computing the c function. In this section, we describe a new branching heuristic called the stochastic gradient descent branching (SGDB) heuristic that works around this issue by cheaply approximating $c : PA \to \{1, 0\}$.

We use online stochastic gradient descent to learn the logistic regression function $\tilde{c} : \mathbb{R}^n \to [0, 1]$ where $\mathbb{R}^n$ is the partial assignment's feature vector and $[0, 1]$ is the probability the partial assignment is in class 1, the conflict class. Online training is a good fit since the function c we are approximating is non-stationary due to the clause database changing over time. For an instance with n Boolean variables and a partial assignment PA, we introduce the features $x_1, ..., x_n$ defined as follows: $x_i = 1$ if variable $i \in PA$, otherwise $x_i = 0$.

Recall that $\tilde{c} := \sigma(w_0 + w_1x_1 + w_2x_2 + ... + w_nx_n)$ is parameterized by the weights w_i, and the goal of SGDB is to find good weights dynamically as the solver roams through the search space. At the start of the search all weights are initialized to zero since we assume no prior knowledge.

To train these weights, SGDB needs to generate training data of the form $PA \times \{1, 0\}$ where 1 signifies the conflicting class, that is, applying BCP on PA with the current clause database causes a conflict. We leverage the existing conflict analysis procedure in the CDCL algorithm to create this data. Whenever the solver performs conflict analysis, SGDB creates a partial assignment PA_1

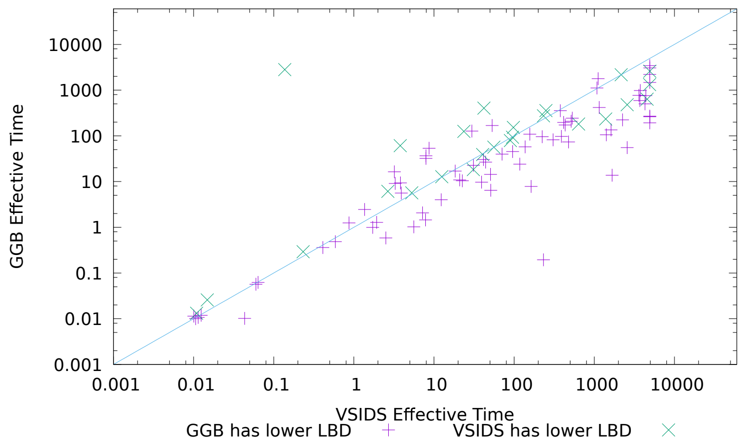

Fig. 2. GGB vs VSIDS. GGB has a lower average LBD for 72 of the 98 comparable instances. GGB has a mean average LBD of 37.2 for this benchmark whereas VSIDS has a mean average LBD of 61.1.

by concatenating the literals on the conflict side of conflict analysis with the negation[3] of the literals in the learnt clause and gives this partial assignment the label 1. Clearly applying BCP to PA_1 with the current clause database leads to a conflict, hence it is assigned to the conflict class. SGDB creates another partial assignment PA_0 by concatenating all the literals in the current partial assignment excluding the variables in the current decision level and excluding the variables in PA_1. Applying BCP to PA_0 does not lead to a conflict with the current clause database, because if it did, the conflict would have occurred at an earlier level. Hence PA_0 is given the label 0. In summary, SGDB creates two data points at every conflict, one for each class (the conflict class and the non-conflict class) guaranteeing a balance between the two classes.

During conflict, two data points are created. SGDB then applies one step of stochastic gradient descent on these two data points to update the weights. Since we are training in an online fashion, the two data points are discarded after the weights are updated. To reduce the computation cost, regularization is performed lazily. Regularization, if done eagerly, updates the weights of every variable on every step of stochastic gradient descent. With lazy updates, only the weights of non-zero features are updated. As is typical with stochastic gradient descent, we gradually decrease the learning rate α over time until it reaches a fixed limit. This helps to rapidly adjust the weights at the start of the search.

When it comes time to pick a new decision variable, SGDB uses the $\tilde{c}$ function to predict the decision variable that maximizes the probability of creating a partial assignment in class 1, the conflict class. More precisely, it selects the following

[3] Recall that the learnt clause is created by negating some literals in the implication graph, this negation here is to un-negate them.

variable: $argmax_{v \in UnassignedVars} \tilde{c}(CPA \cup PhaseSaving(v))$ where CPA is the current partial assignment and $PhaseSaving(v)$ returns v plus the sign which the phase saving heuristic assigns to v if it were to be branched on. However, the complexity of the above computation is linear to the number of unassigned variables. Luckily this can be simplified by the following reasoning:

$$argmax_{v \in UnassignedVars} \tilde{c}(CPA \cup PhaseSaving(v))$$

$$= argmax_{v \in UnassignedVars} \sigma(w_0 + w_v + \sum_{l \in vars(CPA)} w_l)$$

Note that σ is a monotonically increasing function.

$$= argmax_{v \in UnassignedVars} (w_0 + w_v + \sum_{l \in vars(CPA)} w_l)$$

Remove the terms common to all the iterations of argmax.

$$= argmax_{v \in UnassignedVars} w_v$$

Hence it is equivalent to branching on the unassigned variable with the highest weight. By storing the weights in a max priority queue, the variable with the highest weight can be retrieved in time logarithmic to the number of unassigned variables, a big improvement over linear time. The complete algorithm is presented in Algorithm 2.

Differences with VSIDS: The SGDB branching heuristic presented thus far has many similarities with VSIDS. During each conflict, VSIDS increments the activities of the variables in PA_1 by 1 whereas SGDB increases the weights of the variables in PA_1 by a gradient. Additionally, the VSIDS decay multiplies every activity by a constant between 0 and 1, the L2 regularization in stochastic gradient descent also multiplies every weight by a constant between 0 and 1. SGDB decreases the weights of variables in PA_0 by a gradient, VSIDS does not have anything similar to this.

Sparse Non-Conflict Extension: The `AfterConflictAnalysis` procedure in Algorithm 2 takes time proportional to $|PA_1|$ and $|PA_0|$. Unfortunately in practice, $|PA_0|$ is often quite large, about 75 times the size of $|PA_1|$ in our experiments. To shrink the size of PA_0, we introduce the sparse non-conflict extension. With this extension PA_0 is constructed by randomly sampling one assigned literal for each decision level less than the current decision level. Then the literals in PA_1 are removed from PA_0 as usual. This construction bounds the size of PA_0 to be less than the number of decision levels. See Algorithm 3 for the pseudocode.

Reason-Side Extension: SGDB constructs the partial assignment PA_1 by concatenating the literals in the conflict side and the learnt clause. Although PA_1 is sufficient for causing the conflict, the literals on the reason side are the reason why PA_1 literals are set in the first place. Inspired by the LRB branching heuristic with a similar extension, the reason-side extension takes the literals on the reason side adjacent to the learnt clause in the implication graph and adds

Algorithm 2 Pseudocode for the *SGDB* heuristic.

```
1: function PhaseSaving(Var)                                   ▷ return the variable plus a sign
2:     return $mkLit(Var, Var_{SavedPolarity})$
3:
4: procedure Initialize
5:     for all $v \in Vars$ do
6:         $\alpha \leftarrow 0.8$, $\lambda \leftarrow 0.1 \times \alpha$, $w_v \leftarrow 0$
7:         $r_v \leftarrow 0$                                  ▷ Stores the last time $v$ was lazily regularized.
8:     $conflicts \leftarrow 0$                                ▷ The number of conflicts occurred so far.
9:
10: function GetPA1($learntClause$, $conflictSide$)
11:     return $\{\neg l \mid l \in learntClause\} \cup conflictSide$
12:
13: function GetPA0($PA_1$)
14:     return $\{v \in AssignedVars \mid DecisionLevel(v) < currentDecisionLevel\} \setminus PA_1$
15:
16: procedure AfterConflictAnalysis($learntClause$, $conflictSide$)   ▷ Called after a learnt
    clause is generated from conflict analysis.
17:     if $\alpha > 0.12$ then
18:         $\alpha \leftarrow \alpha - 2 \times 10^{-6}$, $\lambda \leftarrow 0.1 \times \alpha$
19:     $conflicts \leftarrow conflicts + 1$
20:     $PA_1 \leftarrow GetPA1(learntClause, conflictSide)$
21:     $PA_0 \leftarrow GetPA0(PA_1)$
22:     for all $v \in vars(PA_1 \cup PA_0)$ do                  ▷ Lazy regularization.
23:         if $conflicts - r_v > 1$ then
24:             $w_v \leftarrow w_v \times (1 - \frac{\alpha\lambda}{2})^{conflicts - r_v - 1}$
25:         $r_v \leftarrow conflicts$
26:     $error_1 \leftarrow \sigma(w_0 + \sum_{i \in vars(PA_1)} w_i)$   ▷ Compute the gradients and descend.
27:     $error_0 \leftarrow \sigma(w_0 + \sum_{i \in vars(PA_0)} w_i)$
28:     $w_0 \leftarrow w_0 \times (1 - \frac{\alpha\lambda}{2}) - \frac{\alpha}{2}(error_1 + error_2)$
29:     for all $v \in vars(PA_1)$ do
30:         $w_v \leftarrow w_v \times (1 - \frac{\alpha\lambda}{2}) - \frac{\alpha}{2}(error_1)$
31:     for all $v \in vars(PA_0)$ do
32:         $w_v \leftarrow w_v \times (1 - \frac{\alpha\lambda}{2}) - \frac{\alpha}{2}(error_0)$
33:
34: function SGDB
35:     $d \leftarrow argmax_{v \in UnassignedVars} w_v$
36:     while $conflicts - r_d > 0$ do                           ▷ Lazy regularization.
37:         $w_d \leftarrow w_d \times (1 - \frac{\alpha\lambda}{2})^{conflicts - r_d}$
38:         $r_d \leftarrow conflicts$
```

them to PA_1. This lets the learning algorithm associate these variables with the conflict class. See Algorithm 4 for the pseudocode.

5.1 Experimental Results

We ran MapleSAT configured with 6 different branching heuristics (LRB, VSIDS, SGDB with four combinations of the two extensions) on all the application and hard combinatorial instances from SAT Competitions 2011, 2013, 2014, and 2016. At the end of each run, we recorded the elapsed time, the GLR at termination, and the average LBD of all clauses learnt from start to finish. Table 3 and Figure 3 show the effectiveness of each branching heuristic in solving the instances in the benchmark. The reason-side extension (resp. sparse non-conflict extension) increases the number of solved instances by 97 (resp. 155). The two extensions together increase the number of solved instances by 219,

Algorithm 3 Pseudocode for the sparse non-conflict extension. Only the `GetPA0` code is modified, the rest remains the same as SGDB.

```
1: function Sample($level$)
2:     $C \leftarrow \{v \in Vars \mid DecisionLevel(v) = level\}$
3:     return a variable sampled uniformly at random from $C$
4:
5: function GetPA0($PA_1$)
6:     return $(\bigcup_{i \in \{1,2,\ldots,currentDecisionLevel-1\}} Sample(i)) \setminus PA_1$
```

Algorithm 4 Pseudocode for the reason-side extension. Only the `GetPA1` code is modified, the rest remains the same as SGDB.

```
1: function GetPA1($learntClause$, $conflictSide$)
2:     $adjacent \leftarrow \bigcup_{lit \in learntClause} Reason(\neg lit)$
3:     return $\{\neg l \mid l \in learntClause\} \cup conflictSide \cup adjacent$
```

and in total solve just 12 instances fewer than VSIDS. LRB solves 93 more instances than VSIDS. Table 4 shows the GLR and the average LBD achieved by the branching heuristics. Both extensions individually increased the GLR and decreased the LBD. The extensions combined increased the GLR and decreased the LBD even further. The best performing heuristic, LRB, achieves the highest GLR and lowest LBD in this experiment. It should not be surprising that LRB has high GLR, our goal when designing LRB was to generate lots of conflicts by branching on variables likely to cause conflicts. By design, LRB tries to achieve high GLR albeit indirectly by branching on variables with high learning rate.

6 Threats To Validity

1. **Did we overfit?** One threat is the possibility that the parameters are overtuned for the benchmarks and overfit them, and hence work poorly for untested benchmarks. To avoid overtuning parameters, we chose $\frac{\alpha}{2}$ in SGD to be the same as the step-size in LRB from our previous paper [19] and also chose $(1 - \frac{\alpha\lambda}{2})$ to be the same as the locality extension penalty factor in LRB from the same paper. We fixed these parameters from the start and never tuned them. Also, note that the training is online per instance.
2. **What about optimizing for quality of learnt clauses?** This remains a challenge. We did notice that when we maximize GLR we get a very nice side-effect of low LBD. Having said that, in the future we plan to explore other notions of quality and integrate that into a multi-objective optimization problem view of branching heuristics.

7 Related Work

The VSIDS branching heuristic, currently the most widely implemented branching heuristic in CDCL solvers, was introduced by the authors of the Chaff solver in 2001 [23] and later improved by the authors of the MiniSat solver in 2003 [11].

Table 3. # of solved instances by various configurations of SGD, VSIDS, and LRB.

Benchmark	Status	SGDB + No Ext	SGDB + Reason Ext	SGDB + Sparse Ext	SGDB + Both Ext	VSIDS	LRB
2011 Application	SAT	84	89	96	93	95	103
	UNSAT	87	87	96	94	99	98
	BOTH	171	176	192	187	194	201
2011 Hard Combinatorial	SAT	85	92	91	97	88	93
	UNSAT	36	50	43	51	48	64
	BOTH	121	142	134	148	136	157
2013 Application	SAT	91	92	108	112	127	132
	UNSAT	75	75	86	81	86	95
	BOTH	166	167	194	193	213	227
2013 Hard Combinatorial	SAT	107	109	118	118	115	116
	UNSAT	57	88	60	99	73	111
	BOTH	164	197	178	217	188	227
2014 Application	SAT	79	86	100	107	105	116
	UNSAT	65	62	79	73	94	76
	BOTH	144	148	179	180	199	192
2014 Hard Combinatorial	SAT	82	82	91	86	91	91
	UNSAT	41	61	56	73	59	89
	BOTH	123	143	147	159	150	180
2016 Application	SAT	52	55	62	62	60	61
	UNSAT	52	50	55	57	63	65
	BOTH	104	105	117	119	123	126
2016 Hard Combinatorial	SAT	5	7	6	7	3	6
	UNSAT	19	29	25	26	42	25
	BOTH	24	36	31	33	45	31
TOTAL (no duplicates)	SAT	585	612	672	682	684	718
	UNSAT	432	502	500	554	564	623
	BOTH	1017	1114	1172	1236	1248	1341

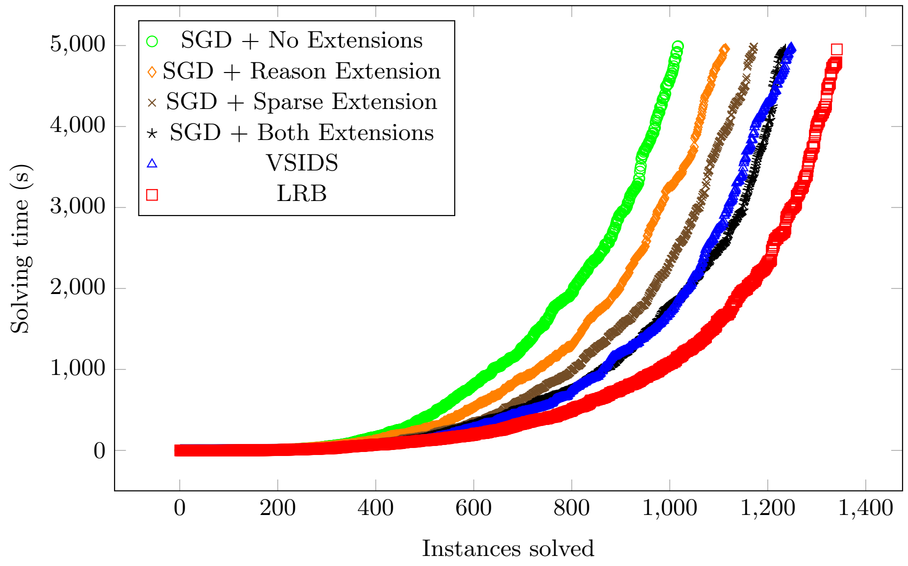

Fig. 3. A cactus plot of various configurations of SGD, VSIDS, and LRB on the entire benchmark with duplicate instances removed.

Table 4. GLR and average LBD of various configurations of SGD, VSIDS, and LRB on the entire benchmark with duplicate instances removed. LRB not solves the most instances and achieves the highest GLR and lowest average LBD in our experiments.

Metric	Status	SGDB + No Ext	SGDB + Reason Ext	SGDB + Sparse Ext	SGDB + Both Ext	VSIDS	LRB
Mean GLR	SAT	0.324501	0.333763	0.349940	0.357161	0.343401	0.375181
	UNSAT	0.515593	0.518362	0.542679	0.545567	0.527546	0.557765
	BOTH	0.403302	0.409887	0.429420	0.434854	0.419337	0.450473
Mean Avg LBD	SAT	22.553479	20.625091	19.470764	19.242937	28.833872	16.930723
	UNSAT	17.571518	16.896552	16.249930	15.832730	22.281780	13.574527
	BOTH	20.336537	18.965914	18.037512	17.725416	25.918232	15.437237

Carvalho and Marques-Silva introduced a variation of VSIDS in 2004 where the bump value is determined by the learnt clause length and backjump size [8] although their technique is not based on machine learning. Lagoudakis and Littman introduced a new branching heuristic in 2001 that dynamically switches between 7 different branching heuristics using reinforcement learning to guide the choice [17]. Liang et al. introduced two branching heuristics, CHB and LRB, in 2016 where a stateless reinforcement learning algorithm selects the branching variables themselves. CHB does not view branching as an optimization problem, whereas LRB, GGB, SGDB do. As stated earlier, LRB optimizes for learning rate, a metric defined with respect to variables. GGB and SGDB optimize for global learning rate, a metric defined with respect to the solver.

8 Conclusion and Future Work

Finding the optimal branching sequence is nigh impossible, but we show that using the simple framework of optimizing GLR has merit. The crux of the question since the success of our LRB heuristic is whether solving the learning rate optimization problem is indeed a good way of designing branching heuristics. A second question is whether machine learning algorithms are the way to go forward. We answer both questions via a thorough analysis of 7 different notable branching heuristics, wherein we provide strong empirical evidence that better branching heuristics correlate with higher GLR. Further, we show that higher GLR correlates with lower LBD, a popular measure of quality of learnt clauses. Additionally, we designed a greedy branching heuristic to maximize GLR and showed that it outperformed VSIDS, one of the most competitive branching heuristics. To answer the second question, we designed the SGDB that is competitive vis-a-vis VSIDS. With the success of LRB and SGDB, we are more confident than ever before in the wisdom of using machine learning techniques as a basis for branching heuristics in SAT solvers.

9 Acknowledgement

We thank Sharon Devasia Isac and Nisha Mariam Johnson from the College Of Engineering, Thiruvananthapuram, for their help in implementing the Berkmin and DLIS branching heuristics.

References

[1] https://sites.google.com/a/gsd.uwaterloo.ca/maplesat/
[2] https://sites.google.com/a/gsd.uwaterloo.ca/maplesat/sgd
[3] Audemard, G., Simon, L.: Predicting Learnt Clauses Quality in Modern SAT Solvers. In: Proceedings of the 21st International Joint Conference on Artificial Intelligence. pp. 399–404. IJCAI'09, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2009)
[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.: Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. FMV Report Series Technical Report 10(1) (2010)
[6] Bottou, L.: On-line Learning in Neural Networks. chap. On-line Learning and Stochastic Approximations, pp. 9–42. Cambridge University Press, New York, NY, USA (1998)
[7] 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. pp. 322–335. CCS '06, ACM, New York, NY, USA (2006)
[8] Carvalho, E., Silva, J.P.M.: Using Rewarding Mechanisms for Improving Branching Heuristics. In: SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings (2004)
[9] Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded Model Checking Using Satisfiability Solving. Formal Methods in System Design 19(1), 7–34 (2001)
[10] Cox, D.R.: The Regression Analysis of Binary Sequences. Journal of the Royal Statistical Society. Series B (Methodological) 20(2), 215–242 (1958)
[11] Eén, N., Sörensson, N.: Theory and Applications of Satisfiability Testing: 6th International Conference, SAT 2003, Santa Margherita Ligure, Italy, May 5-8, 2003, Selected Revised Papers, chap. An Extensible SAT-solver, pp. 502–518. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
[12] Fisher, R.A.: Frequency Distribution of the Values of the Correlation Coefficient in Samples from an Indefinitely Large Population. Biometrika 10(4), 507–521 (1915)
[13] Goldberg, E., Novikov, Y.: BerkMin: A Fast and Robust Sat-solver. Discrete Appl. Math. 155(12), 1549–1561 (Jun 2007)
[14] Jeroslow, R.G., Wang, J.: Solving Propositional Satisfiability Problems. Annals of Mathematics and Artificial Intelligence 1(1-4), 167–187 (Sep 1990)
[15] Katebi, H., Sakallah, K.A., Marques-Silva, J.a.P.: Empirical Study of the Anatomy of Modern Sat Solvers. In: Proceedings of the 14th International Conference on Theory and Application of Satisfiability Testing. pp. 343–356. SAT'11, Springer-Verlag, Berlin, Heidelberg (2011)
[16] Kautz, H., Selman, B.: Planning As Satisfiability. In: Proceedings of the 10th European Conference on Artificial Intelligence. pp. 359–363. ECAI '92, John Wiley & Sons, Inc., New York, NY, USA (1992)
[17] Lagoudakis, M.G., Littman, M.L.: Learning to Select Branching Rules in the DPLL Procedure for Satisfiability. Electronic Notes in Discrete Mathematics 9, 344–359 (2001)
[18] Liang, J.H., Ganesh, V., Poupart, P., Czarnecki, K.: Exponential Recency Weighted Average Branching Heuristic for SAT Solvers. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence. pp. 3434–3440. AAAI'16, AAAI Press (2016)

[19] Liang, J.H., Ganesh, V., Poupart, P., Czarnecki, K.: Learning Rate Based Branching Heuristic for SAT Solvers. In: Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings. pp. 123–140 (2016)

[20] Luby, M., Sinclair, A., Zuckerman, D.: Optimal Speedup of Las Vegas Algorithms. Information Processing Letters 47(4), 173–180 (Sep 1993)

[21] Marques-Silva, J.P.: The Impact of Branching Heuristics in Propositional Satisfiability Algorithms. In: Proceedings of the 9th Portuguese Conference on Artificial Intelligence: Progress in Artificial Intelligence. pp. 62–74. EPIA '99, Springer-Verlag, London, UK, UK (1999)

[22] 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. pp. 220–227. ICCAD '96, IEEE Computer Society, Washington, DC, USA (1996)

[23] 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. pp. 530–535. DAC '01, ACM, New York, NY, USA (2001)

[24] Murphy, K.P.: Machine Learning: A Probabilistic Perspective. The MIT Press (2012)

[25] Pipatsrisawat, K., Darwiche, A.: A Lightweight Component Caching Scheme for Satisfiability Solvers. In: Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing. pp. 294–299. SAT'07, Springer-Verlag, Berlin, Heidelberg (2007)

[26] Soos, M.: CryptoMiniSat v4. SAT Competition p. 23 (2014)

[27] Spearman, C.: The Proof and Measurement of Association between Two Things. The American Journal of Psychology 15(1), 72–101 (1904)

[28] Stump, A., Sutcliffe, G., Tinelli, C.: 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, chap. StarExec: A Cross-Community Infrastructure for Logic Solving, pp. 367–373. Springer International Publishing, Cham (2014)