Machine Learning for SAT Solvers

by

Jia Hui Liang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2018

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Karem Sakallah Professor, Electrical Engineering and Computer Science Dept., University of Michigan
Supervisors:	Vijay Ganesh Assistant Professor, Dept. of Electrical and Computer Engineering, University of Waterloo
	Krzysztof Czarnecki Professor, Dept. of Electrical and Computer Engineering, University of Waterloo
Internal Member:	Derek Rayside Associate Professor, Dept. of Electrical and Computer Engineering, University of Waterloo
Internal-External Member:	Pascal Poupart Professor, Dept. of Computer Science, University of Waterloo
Other Member:	Catherine Gebotys Professor, Dept. of Electrical and Computer Engineering, University of Waterloo

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Boolean SAT solvers are indispensable tools in a variety of domains in computer science and engineering where efficient search is required. Not only does this relieve the burden on the users of implementing their own search algorithm, they also leverage the surprising effectiveness of modern SAT solvers. Thanks to many decades of cumulative effort, researchers have made persistent improvements to SAT technology to the point where nowadays the best solvers are routinely used to solve extremely large instances with millions of variables. Even though our current paradigm of SAT solvers runs in worst-case exponential time, it appears that the techniques and heuristics embedded in these solvers avert the worst-case exponential time in practice. The implementations of these various solver heuristics and techniques are vital to the solvers effectiveness in practice.

The state-of-the-art heuristics and techniques gather data during the run of the solver to inform their choices like which variable to branch on next or when to invoke a restart. The goal of these choices is to minimize the solving time. The methods in which these heuristics and techniques process the data generally do not have theoretical underpinnings. Consequently, understanding why these heuristics and techniques perform so well in practice remains a challenge and systematically improving them is rather difficult. This goes to the heart of this thesis, that is to utilize machine learning to process the data as part of an optimization problem to minimize solving time. Research in machine learning exploded over the past decade due to its success in extracting useful information out of large volumes of data. Machine learning outclasses manual handcoding in a wide variety of complex tasks where data are plentiful. This is also the case in modern SAT solvers where propagations, conflict analysis, and clause learning produces plentiful of data to be analyzed, and exploiting this data to the fullest is naturally where machine learning comes in. Many machine learning techniques have a theoretical basis that makes them easy to analyze and understand why they perform well.

The branching heuristic is the first target for injecting machine learning. First we studied extant branching heuristics to understand what makes a branching heuristics good empirically. The fundamental observation is that good branching heuristics cause lots of clause learning by triggering conflicts as quickly as possible. This suggests that variables that cause conflicts are a valuable source of data. Another important observation is that the state-of-the-art VSIDS branching heuristic internally implements an exponential moving average. This highlights the importance of accounting for the temporal nature of the data when deciding to branch. These observations led to our proposal of a series of machine learning-based branching heuristics with the common goal of selecting the branching variables to increase probability of inducing conflicts. These branching heuristics are shown

empirically to either be on par or outcompete the current state-of-the art.

The second area of interest for machine learning is the restart policy. Just like in the branching heuristic work, we first study restarts to observe why they are effective in practice. The important observation here is that restarts shrink the assignment stack as conjectured by other researchers. We show that this leads to better clause learning by lowering the LBD of learnt clauses. Machine learning is used to predict the LBD of the next clause, and a restart is triggered when the LBD is excessively high. This policy is shown to be on par with state-of-the-art. The success of incorporating machine learning into branching and restarts goes to show that machine learning has an important role in the future of heuristic and technique design for SAT solvers.

Acknowledgements

I would like to thank my co-supervisor **Vijay Ganesh** for his wisdom and guidance in academics, career, and life. Your ability to teach not just knowledge but also character has profoundly changed me as a person. We make a great research team and I am sincerely fortunate to have your mentorship for the last half decade.

I would like to thank my other co-supervisor **Krzysztof Czarnecki** for his flexibility and openness for me to find my place as a budding researcher. I owe a great deal of debt for your kindness and patience over the years.

I would like to thank the members of the Examining Committee for their time and expertise. Their advices are invaluable for improving the quality of this thesis.

Dedication

This is dedicated to ma and pa.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Boolean satisfiability problem, one of the most fundamental one in computer science, asks if there exists an (efficient) decision procedure that decides whether Boolean formulas given in conjunctive normal form are satisfiable. Given that Boolean satisfiability (aka SAT) is the quintessential NP-complete problem [29], the broad consensus is that there are no efficient decision procedures or solvers for it. This consensus might lead one to prematurely conclude that SAT solvers cannot be useful for any real-world applications and that this problem is primarily of theoretical interest. Yet nothing could be further from the truth. Modern SAT solvers have had a revolutionary impact on a wide range of disciplines. These solvers routinely solve very large instances, with tens of millions of variables and clauses in them, obtained from real-world applications. Through decades of research, the community of solver researchers has built surprisingly effective backtracking algorithms called conflict-driven clause-learning (CDCL) SAT solvers [66] that are based on just a handful of key principles [52]: conflict-driven branching, efficient propagation, conflict analysis and conflict-driven clause-learning, preprocessing/inprocessing, and restarts. What is even more surprising is that these solvers scale well for many applications for which they are not designed, often out-performing the previous best algorithms that are specifically designed for the domain. Examples include contexts as diverse as program analysis [24], model-checking [26, 21], AI planning [54], cryptanalysis [70], and combinatorial mathematics [56, 46, 90].

Despite all this success, a persistent issue with solver research has been a lack of good empirical and theoretical understanding of why these solvers are so efficient. Such an understanding is crucial not merely due to its scientific merit, but also because it would enable us to build better solvers going forward. The current trial-and-error approach of improving solver performance is not sustainable in the long run. We need general principles

to guide researchers as they navigate the complex space of solver heuristics. In this thesis, we address this question of what constitute good general principles for solver heuristic design.

In this thesis, we approach the knotty problem of why solvers are efficient by empirically studying the most successful solver heuristics, such as branching and restarts, in detail. This systematic study led to some interesting and surprising observations. The first among them was the rather obvious observation that solver heuristics are methods aimed at optimizing some metric that correlates well with minimizing solver runtime. Further, we noticed that some of these successful heuristics, especially those used in branching, leveraged the large amount of data generated by solvers via appropriate statistical analysis. Finally, as we studied them in detail, we realized that solver designers had inadvertently incorporated concepts from machine learning, such as exponential moving averages (EMA) and exponential recency weighted averages (ERWA) used in the context of reinforcement learning. This led us to the following thesis:

> An effective and general principle for designing SAT solvers (or any logic engine) is to view these decision procedures as a collection of interacting optimization subroutines, where each subroutine or heuristic aims at optimizing some metric that correlates with minimizing overall solver runtime on any given input instance. While identifying such metrics is hard and may require knowledge of various application domains, they do dramatically clarify solver design. Further, these subroutines have access to the enormous amount of data that solvers generate, which in turn suggests the use of machine learning (ML) based methods to designing them. Finally, as we briefly describe below and at length in the rest of the thesis, online machine learning methods are particularly well-suited for designing efficient solver heuristics.

While there are many kinds of SAT solvers, in this thesis we focus on *conflict-driven clause-learning* (CDCL) SAT solvers [66] since they are presently the dominant solver for most practical applications. As is the case with any backtracking search algorithm, the run of a CDCL SAT solver can be visualized as traversing paths over a search tree. In this search tree, each node is a distinct variable with two outgoing edges marked true and false respectively denoting assigning values to the variables. The branching heuristic decides where the variables are placed in this search tree, and this ordering has an enormous impact on the running time of the solver. At every node, the solver infers the values of some variables forced by the value assignments in the nodes above. As the solver traverses down the search tree, it will frequently discover that the path it is exploring contains no

solutions, hence it needs to backtrack and try another branch. Once the solver exhaustively searches the entire search tree without finding a satisfying solution, it terminates having proven the formula unsatisfiable. The solver described thus far is called DPLL after the inventors Davis-Putnam-Logemann-Loveland [31], first presented in 1960. The brute force nature of DPLL limits its practical utility. The CDCL paradigm extends DPLL with a series of techniques and heuristics such as conflict-driven branching, clause learning, backjumping, and frequent restarts that dramatically improve its performance over the plain DPLL solver. See Section 2.1 for a thorough explanation of how CDCL solvers are implemented.

The crucial insight behind the success of CDCL over DPLL is that CDCL combines search (in the form of branching and propagation) with deduction (in the form of conflict analysis and clause learning) in a corrective feedback loop, while DPLL is primarily a search algorithm. The deductive component of the CDCL solver performs two critical roles: first, it prune the search space through deductively eliminating paths in the search tree (via clause learning), and second, it guides the branching via conflict-driven branching heuristic (a la VSIDS). This back-and-forth exploration via search and exploitation by learning clauses aligns very well with certain reinforcement learning paradigms. In the typical reinforcement learning setting, an agent iteratively interacts with an environment with the goal of learning a policy that maximizes its reward. One could view many CDCL solver heuristics, such as branching, restarts, and value selection through this lens, enabling us to get a deeper understanding of why SAT solvers are efficient and further leverage the vast literature on reinforcement learning techniques that might be particularly suited to the SAT solver setting.

The core focus of this thesis is designing new state-of-the-art heuristics for CDCL SAT solvers and it covers two broad themes. The first theme is a detailed empirical understanding of why certain SAT solver heuristics, such as branching and restarts, work so well for real-world instances. The second theme is leveraging this understanding into a general solver design principle that views these heuristics as ML-based optimization procedures. Below we provide a brief description the contributions presented in this thesis.

1.1 Summary of Contributions

Understanding the Power of Heuristics: One of the struggles to improving heuristics is that we lack insight into why they are effective in the first place. Modern CDCL SAT solvers are complex beasts with many convoluted interactions between its various parts. But before we understand the solver in its entirety, we should start by

understanding the individual parts such as the heuristics. Even though the heuristics play an outsized role in the performance of modern SAT solvers, our understanding on why they are effective is lagging behind. In this thesis, we try to better our understanding of why modern implementations of branching heuristics and restart policies are effective in practice. We find that the explanation in both cases relates to clause learning, the defining feature of CDCL SAT solvers. Ultimately, CDCL SAT solvers are proof systems and the branching heuristic and restart policies enhance their ability to produce proofs effectively by enhancing clause learning.

In Chapter 4, we focus on branching heuristics. Every branching heuristic devises some scheme for ranking variables and selects the highest ranked unassigned variable to branch on next. Many great heuristics have been proposed and employed without explicitly answering the question "what makes a variable good to branch on?" We empirically study a large set of extant branching heuristics to give us clues on how to address this question. Our study of extant branching heuristics suggests that better heuristics tend to trigger clause learning more frequently, a metric we call the *global learning rate* (GLR). In other words, a variable is good to branch on if it is likely to lead to a conflict so a new clause can be learnt to prune the search space. We give special attention to the dominant branching heuristic called *variable state independent decaying sum* (VSIDS) [68] and obtain other valuable insights. We show that the decay of VSIDS corresponds to an exponential moving average highlighting the importance of temporality when processing data. Additionally, we show that VSIDS correlates with graph centrality and bridge variables in the community structure. The wisdom we gained from this study motivates the design of our machine learning-based branching heuristics.

Restart policies are the focus of Chapter 5. Just like the case with branching heuristics, we find that restart policies positively affect clause learning. We show that the frequency of restarts negatively correlates with the size of the assignment stack. Restarts allow the solver to reassemble the assignment stack on the subset variables that the branching heuristic is focused on now. In other words, the variables on the assignment stack that are no longer ranked highly by the branching heuristic can be removed by restarts thus shrinking the assignment stack. In turn, a shrinking assignment stack improves the quality of clause learning. Since the assignment stack is more focused with restarts, the learnt clauses are focused on fewer blocks of literals, thus decreasing the *literal block distance* (LBD) [10]. The LBD is a metric for learnt clauses that is widely believed to be negatively correlated with the quality of the clause. These observations about restarts motivate the formation of our machine learning-based restart policy.

Optimization with Machine Learning: CDCL SAT solver heuristics have one goal, solve the instance as quickly as possible. For any problem, there exists a shortest proof of unsatisfiability or satisfiability. It is rarely easy for a CDCL SAT solver to construct the shortest proof directly. Instead the solver relies on a handful of heuristics to make an assortment of choices to hopefully produce a proof with the least amount of time as possible. We believe that we should attack heuristic design from first principles, that is, to explicitly view heuristics as an optimization problem with the objective to minimize total solving time. This optimization is difficult to do so directly since the solving time is hard to compute a priori. Instead, we opt to optimize some other feature that correlates with solving time and also agrees with experience and intuition. Even with this simplification, we still find ourselves running into trouble. As we designed branching heuristics and restart policies based on our observations, we encounter a problem where we need a complicated function as the basis of the heuristic to perform the optimization. The first option is to handcode these complicated functions, but this would either lead a function that is too expensive to invoke or a function that is too cryptic for humans to formulate as computer code. We sidestepped these issues by relying on machine learning to learn the function from the data which happens to be abundant in CDCL SAT solvers.

In Chapter 3, we use the lessons learnt from studying branching heuristics to design new state-of-the-art branching heuristics based on machine learning. The objective we chose is to maximize GLR informed by our earlier study of extant branching heuristics. We first designed the *greedy GLR branching* (GGB) heuristic as a proof of concept to observe the consequence of branching on variables that maximize GLR greedily. The overhead of the GGB is extremely high, but if the branching overhead is factored out, we find that the GGB far exceeds the current state-of-the-art in both maximizing GLR and minimizing solving time. This result reinforces the proposition that GLR is a good objective to optimize for branching heuristics, subject to the overhead. The obstacle with GGB is its overhead, in particular, it calls upon an extremely expensive function to test which variable increases the GLR as the foundation of its GLR maximization. We propose the *stochastic gradient descent branching* (SGDB) heuristic which operates the same as GGB except the expensive function is approximated via supervised machine learning. SGDB is shown to be very competitive with modern state-of-the-art branching heuristics. Additionally, we provided an alternative objective of learning rate with respect to a variable. We used multi-armed bandits, a subset of reinforcement learning, to maximize this variable-specific learning rate. The new heuristic called the *learning rate branching* (LRB) heuristic is shown to solve more instances than the state-of-the-art.

Our observations about restarts inspire our machine learning-based restart policy in Chapter 5. We chose the objective of minimizing LBD based on our study. The objective of the new restart policy is to avoid learning high LBD clauses since these clauses are of lower quality. The strategy is to invoke a restart if the upcoming learnt clause has excessively high LBD. We use supervised machine learning to predict the LBD of the learnt clause using past LBDs as data. We quantify the threshold for "excessively high" by approximating the right tail distribution of LBDs as a normal distribution and use z-scores on this distribution to predict high percentiles. The new restart policy called *machine learning-based restart* (MLR) policy is competitive with state-of-the-art.

Branching heuristics and restart policies are the focus of this thesis as they are among the most important solver heuristics. Based on these two case studies, we believe that the core message of this thesis lifts to other contexts such as clause deletion, clause learning, and polarity selection. That is, a good general strategy for designing heuristics is to first study and understand the power behind heuristics. Secondly, solve the optimization problem of minimizing the solving time, or another feature as a proxy, using machine learning.

While the use of machine learning in SAT solvers is not new, our approach to the problem differs from previous attempts in many important ways. CDCL SAT solvers have many parameters from the various heuristics and techniques it implements, and fine-tuning these parameters to the instance leads to better performance. Various researchers have used machine learning to predict the best parameters by training offline on a large benchmark of instances [49, 64, 57], also known as *algorithm configuration*. The idea is that instances with similar features (for example, similar number of variables and clauses) will likely benefit from similar parameters. Supervised machine learning is used to either predict the best configuration given the instance features or predict the solving time given a configuration. Another prominent use of machine learning in SAT research is in the *portfolio* approach to parallel SAT [87]. A portfolio solver contains many SAT solvers, and runs a small selection of them for a given input with the hope that at least one solver terminates quickly. Since the amount of computer cores on a machine is limited, the portfolio solver must choose a subset among the solvers to run. A portfolio solver prefers a portfolio of diverse solvers so that they can cover each others' weaknesses. Supervised machine learning is used to predict which solvers are most likely to terminate quickest on the given input, and hence those solvers are the ones used. Just like in algorithm configuration, the training is performed offline on a large benchmark of instances. In these use cases, machine learning is external to the CDCL SAT solver itself. These methods treat the SAT solver as a black box. Our approach in this thesis replaces fundamental parts inside the CDCL SAT solver with machine learning. By

changing the solver itself, we can make improvements that black box approaches cannot. Performing machine learning internally in the solver has the major advantage of having access to the data dynamically generated during the run of the solver. This data gives more powerful insights into the input formula than static analysis alone. Additionally, the machine learning training switches from offline to online. The consequence of our approach is that the machine learning technique needs to be extremely computationally efficient when performed online since every second spent on machine learning is a second spent not performing CDCL SAT solving. On the other hand, online training prevents a certain class of overfitting. The offline training uses a large benchmark with the assumption that the learning generalizes to the input instance. Whereas in online training, we train on the input instance itself and need not worry about generalizability between different instances. This allows he heuristic to adapt itself dynamically to the instance being solved. We believe our approach to machine learning is complimentary to the typical uses of machine learning in SAT solvers. Our heuristics, like most solutions based on machine learning, introduce a handful of parameters subject to tuning by automatic algorithm configuration. Additionally, our heuristics veer from the standard heuristics, giving CDCL SAT solvers more diversity thus benefiting portfolio solvers.

Chapter 2

Background

This chapter reviews background knowledge required for the remainder of this thesis, covering the 5 topics: SAT solving in Section 2.1, averages in Section 2.2, probability distributions in Section 2.3, machine learning in Section 2.4, and graph community structure and centrality in Section 2.5.

2.1 SAT Solving

Boolean satisfiability is the problem of finding an assignment to the variables of a Boolean formula such that the formula evaluates to true. SAT solvers are programs that take a Boolean formula as input (denoted by ϕ) and outputs an assignment satisfying the input formula if such an assignment exists, in which case the input formula is said to be *satisfiable*. Otherwise the input formula is said to be *unsatisfiable*. In this section, we describe how SAT solvers work internally including the format of inputs, heuristics and techniques involved, and the backtracking search.

2.1.1 Conjunctive Normal Form (CNF)

Almost all SAT solvers expect the input Boolean formula to be in *conjunctive normal form* (CNF). Any propositional Boolean formula can be transformed into CNF with only a linear increase in formula size using the Tseitin transformation [84]. Hence, requiring CNF as input does not limit the space of allowable Boolean formulas.

A *Boolean variable* is a variable that can take on either the value `true` or the value `false`, typically represented by the letters x, y, or z. A *literal* is either a variable (x) or its negation ($\neg x$), also interpreted as a pair of variable (x) and sign (negative meaning $\neg$ and positive meaning absence of $\neg$). A *clause* is a disjunction ("or") of literals. A *conjunctive normal form* (CNF) formula is a conjunction ("and") of clauses. The length of a clause C, represented by a pair of vertical bars $|C|$, is the number of literals in the clause. To simplify notation, a clause is sometimes treated as a set of literals and a CNF formula is sometimes treated as a set of clauses. Sometimes a clause is treated as a set of variables when the sign is not important. The advantage of CNF is that the SAT solver only needs to worry about clauses and as we will see in the next section, modern SAT solvers implement propagation and clause learning techniques that operates efficiently over clauses.

2.1.2 Conflict-Driven Clause-Learning (CDCL) SAT Solver

Over the years, researchers have invented many paradigms for implementing SAT solvers with various advantages and disadvantages. The conflict-driven clause-learning (CDCL) [66] style SAT solvers is the focus of this thesis as they are presently the most widely used in practice. CDCL SAT solvers are backtracking algorithms that explore the assignment space. At a high level, it tries to constructively build a complete satisfying assignment, backtracking and fixing mistakes as needed.

An *assignment* (denoted by ν) is a mapping of variables to either `true` or `false`. For example $\nu = \{x \mapsto true, y \mapsto false\}$ is an assignment that maps x to true and y to false. A variable is *assigned* if it is in the mapping, otherwise it is *unassigned*. An assignment is *complete* if it maps every variable to some value, otherwise it is *partial*. For convenience, a literal is interchangeable with a variable plus value assignment. That is, $x \equiv x \mapsto true$ and $\neg x \equiv x \mapsto false$. Given an assignment, we can evaluate variables, literals, clauses, and CNFs using the following formulas.

$$EvaluateVariable(x, \nu) := \nu[x] \tag{2.1}$$

$$EvaluateLiteral(l, \nu) := \begin{cases} EvaluateVariable(x, \nu) & \text{if } l = x \\ \neg EvaluateVariable(x, \nu) & \text{if } l = \neg x \end{cases} \tag{2.2}$$

$$EvaluateClause(C, \nu) := \bigvee_{l \in C} EvaluateLiteral(l, \nu) \tag{2.3}$$

$$EvaluateCNF(\phi, \nu) := \bigwedge_{C \in \phi} EvaluateClause(C, \nu) \tag{2.4}$$

A clause is *satisfying* with respect to an assignment (the assignment is usually understood from context) if at least one literal in the clause evaluates to true since the clause is a disjunction. A clause is *falsified* with respect to an assignment if all the literals in the clause evaluate to false. A complete assignment is *satisfying* if all the clauses in the input CNF are satisfied. In other words, the input CNF evaluates to true with a satisfying complete assignment. Any partial assignment that falsifies a clause cannot be extended to a satisfying complete assignment. In a backtracking search like CDCL, a clause being falsified means the solver needs to backtrack.

A CDCL SAT solver relies on a handful of features to help it search for a satisfying complete assignment. We describe the most prominent features below and the entire CDCL algorithm is presented in Algorithm 1.

Assignment: A CDCL SAT solver maintains an assignment, sometimes referred to as the *current partial assignment* or the *assignment stack*, in which it attempts to extend into a complete satisfying assignment. Branching and propagation extend this assignment, whereas backjumping and restarts shrink this assignment.

Branching: The solver *branches* by selecting an unassigned variable, guessing its value, and adding it to the assignment. The *branching heuristic* is the piece of code that selects the variable and value. Typically, the branching heuristic is divided into two parts: the *variable selection heuristic* (typically VSIDS [68]) is responsible for selecting which variable to branch on and the *polarity heuristic* (typically phase-saving [74]) is responsible for selecting the value. In this thesis, we always use phase-saving as the default polarity heuristic and occasionally refer to the variable selection heuristic as the branching heuristic when it is paired with phase-saving. Hence VSIDS is often referred to as a branching heuristic despite only selecting the variable and leaving phase-saving to select the value. This linguistic short cut is very common in the SAT community.

A variable assigned by branching is called a *decision variable*. A decision variable plus its assigned value is a *decision literal*. The *decision level* of a decision variable x is the number of decision variables already in the assignment when x was assigned by branching, not including x itself. The decision level of a literal is the decision level of its variable.

Boolean Constraint Propagation (BCP): A clause of length n is unit with respect to an assignment if $n-1$ literals evaluate to false and the last literal is unassigned. Since a CNF can only be satisfied if this last literal is true, Boolean constraint propagation

(BCP) searches for unit clauses and extends the assignment by assigning the last literal, where the last literal is said to be *propagated*. For example, suppose we have a clause $C = x \vee \neg y \vee \neg z$ and an assignment $x \mapsto false, y \mapsto true$. Since the literals x and $\neg y$ evaluate to false under the assignment, BCP extends the assignment to $x \mapsto false, y \mapsto true, z \mapsto false$ in order to satisfy the clause C. BCP repeats until a fixed-point is reached, that is, no clauses are unit. The *decision level* of a propagated variable x is the number of decision variables already in the assignment when x was propagated.

BCP is the workhorse of a CDCL solver. Even clause learning, the namesake of CDCL, derives its power by empowering BCP. Whenever a variable is propagated, it cuts the current search space in half since the solver does not need to explore the other value for that propagated variable. Since it takes $n - 1$ literals evaluating to false for BCP to kick in, intuitively shorter clauses are better at cutting the search space since they are more likely to propagate.

BCP is responsible for maintaining a data structure called the *implication graph*. The vertices in the implication graph are literals and edges keep track of which literals forced other literals to propagate. For example, in the last example where we have $C = x \vee \neg y \vee \neg z$ and an assignment $x \to false, y \to true$, BCP propagated the literal z. BCP adds and edge from $\neg x$ to z and another edge from y to z since x being false and y being true forced BCP to propagate z. Additionally, falsified clauses have edges from the negation of each of its literal to a special vertex $\perp$. For example, if the clause $x \vee \neg y$ is falsified, then there is an edge from $\neg x$ to $\perp$ and y to $\perp$. Literals corresponding to decision variables do not have incoming edges. The implication graph is used by clause learning, which is described next.

Clause Learning: If a clause is falsified after BCP terminates, then the solver is in *conflict* and the falsified clause is called the *conflicting clause*. The defining feature of CDCL is to learn a new clause by *conflict analysis* after each conflict that succinctly summarizes why the conflict occurred. These are called *learnt clauses* and no satisfying complete assignment can falsify them. In other words, learnt clauses are implied by the original formula. There are various ways to construct a valid learnt clause from a conflict. The learnt clause is stored in the *learnt clause database*. BCP uses all the clauses in the original input and the clauses in the learnt clause database to perform propagation.

The set of all current decision literals during a conflict clearly cannot be all simultaneously be true since it led to the current conflict. That is, at least one of the decision literals must be false. The simplest method to learn a clause during a conflict is to

disjunct the negation of every current decision literal. For example, suppose the solver encounters a conflict and the decision literals are x, $\neg y$, and z. Then a valid learnt clause is $\neg x \vee y \vee \neg z$. As we can see, the learnt clause forces at least one of the decision literals to take the opposite value. This is a valid learnt clause because it is *1-provable* [75].

Definition 1 (1-Provable) *Let C be a clause. Create an assignment A consisting of the negations of all the literals in C, that is $A = \{\neg l \mid l \in C\}$. The clause C is 1-provable if and only if applying BCP with A on the CNF ϕ results in a conflict. In other words, C must be true or else ϕ will have a falsified clause.*

The term 1-provable comes from the fact that the clause C is proved by a single step of BCP. 1-provability is sufficient, but not necessary, for a clause to be a valid learnt clause.

As discussed earlier, a 1-provable learnt clause can be constructed by disjuncting the negation of every current decision literal. However, this creates an unnecessarily large clause and as intuited earlier, smaller clauses are preferable. A general technique for constructing a 1-provable learnt clause is to *cut* the implication graph. Imagine drawing the implication graph on a 2-D plane, then a cut is a contiguous line that bisects the graph into two sides such that one side contains the vertex $\perp$ (called the *conflict side*) and the other side contains all the decision literals (called the *reason side*). Then disjuncting the negation of all the reason side literals incident to the cut is a 1-provable learnt clause. To see why this learnt clause is 1-provable, assigning all these literals then applying BCP will reconstruct the same conflict side leading to the $\perp$, hence a clause is falsified and thus the constructed learnt clause is 1-provable.

The remaining question is where to place the cut in the implication graph. In practice, the cut is placed at the *first unique implication point* (1-UIP) [66]. A *unique implication point* (UIP) is a literal in the implication graph such that all paths starting from the decision literal with the highest decision level to the vertex $\perp$ must cross the UIP vertex. Multiple literals can be a UIP, the 1-UIP refers to the UIP closest to $\perp$. All SAT solvers discussed in this thesis construct the learnt clause by cutting the implication graph such that 1-UIP is on the reason side incident to the cut. The learnt clause contains exactly one literal with the highest decision level, that being the 1-UIP literal itself. There are a couple advantages with cutting at the 1-UIP to learn a clause: the learnt clause is unit after backjumping and the 1-UIP learnt clause has the lowest literal block distance amongst all the UIP learnt clauses. Backjumping and literal block distance will be described shortly.

Learnt clauses are useful because they prune the search space. To state this more formally, learnt clauses are *1-empowering* [75].

Definition 2 (1-Empowering) *Let $C = \alpha \vee l$ be a clause where α is a disjunction of literals and l is a single literal. l is an empowering literal of C with respect to CNF ϕ if and only if the following properties are true.*

1. *The clause C is logically implied by ϕ, that is, all assignments satisfying ϕ also satisfies C.*
2. *Create an assignment A consisting of the negations of all the literals in α, that is $A = \{\neg m \mid m \in \alpha\}$. Applying BCP with A on the CNF ϕ results in a conflict. Additionally, BCP does not propagate l, that is l remains unassigned after BCP terminates.*

The clause C is 1-empowering if it contains at least one empowering literal.

The second property of 1-empowering describes how learnt clauses improve the BCP. Without the learnt clause C, BCP was not able to deduce l. After the learnt clause C is added to the learnt clause database, then it is able to deduce l using C. Therefore propagation can deduce new facts because C is 1-empowering. All learnt clauses produced by UIP learning is 1-empowering.

The clause learnt by cutting the implication graph can also be constructed by applying resolution to the clauses on the conflict side. Resolution is a rule of inference for propositional logic that works as follows.

$$\frac{a_1 \vee ... \vee a_n \vee l \qquad b_1 \vee ... \vee b_m \vee \neg l}{a_1 \vee ... \vee a_n \vee b_1 \vee ... \vee b_m} \; \textit{Resolution}$$

a_i, b_i, and l are literals. Resolution takes two clauses as input with the condition that they share one variable with opposing signs (i.e., l and $\neg l$). The output of resolution, also called the *resolvent*, is a new clause that disjuncts the two input clauses and removes l and $\neg l$. A proof system containing just the resolution inference rule, called *general resolution*, is both sound and complete. As stated earlier, the clause learning technique implemented by CDCL SAT solvers is essentially applying the resolution rule. That is, we can apply the resolution rule to the clauses on the conflict side of the implication graph to construct the exact same learnt clause as a CDCL SAT solver. A CDCL SAT solver returns unsatisfiable when a conflict occurs at decision level zero,

and this is equivalent to inferring an empty clause as a resolvent of the resolution rule. By this perspective, a CDCL solver can be viewed as a resolution engine constructing a proof of the empty clause for an unsatisfiable CNF input. So a typical CDCL solver cannot be stronger than general resolution. Remarkably, Pipatsrisawat and Darwiche showed in 2009 that CDCL solvers with nondeterministic branching and restarts are only at most polynomially weaker than general resolution [75], that is CDCL p-simulates general resolution. We find it beneficial to view SAT solvers with the viewpoint of proof construction, especially when it comes to designing heuristics.

Backjump: After a conflict, the solver needs to undo part of the assignment since no extensions of that assignment are satisfying. Naively, a solver can simply *backtrack* by unassigning all the variables with the highest decision level, that is going from n decision levels to $n-1$ decision levels. In practice however, CDCL solvers typically backtrack multiple decision levels also called *backjump*, also known as a non-chronological backtrack [66]. The rule implemented in most CDCL solvers is to backjump to the smallest decision level that the newest learnt clause becomes unit. For a learnt clause of length 1, that is decision level 0. Otherwise for a learnt clause produced by a UIP cut, the solver backjumps to the second highest decision level of all the variables in the learnt clause since a UIP cut guarantees exactly one variable in the learnt clause with the highest decision level. When backjumping unassigns variables, their corresponding vertices in the implication graph are deleted in unison.

Restart: A restart [43] unassigns all variables in the current assignment and clears the entire implication graph. Semantically, a restart is equivalent to a backjump to decision level 0. Whenever BCP terminates without a conflict, a solver can opt to perform a restart, typically guided by some heuristic. Restarts appear counterintuitive by discarding all the work building the assignment, yet in practice a solver that implements restarts is much more efficient than a solver that does not. This phenomenon is the subject of Chapter 5.

Clause Deletion: A CDCL SAT solver learns a new clause for each conflict that needs to be stored in the learnt clause database. The solver benefits from these learnt clauses by pruning the search space via 1-empowerment but this comes with drawbacks. First, the amount of learnt clauses eventually overwhelms the solver's available memory. Second, the computational cost of BCP increases as the number of learnt clauses increases. For both these reasons, modern CDCL solvers perform *clause deletion* where a large portion of learnt clauses are routinely deleted. The downside is that the solver may need to relearn some deleted clauses. If the deletion is too aggressive, then the solver can get stuck in an infinite loop of relearning the same clauses

over and over, thus the solver is no longer guaranteed to terminate. In practice, the frequency of clause deletion decreases over time to alleviate this issue.

To mitigate the drawbacks of clause deletion, solvers try to delete lower "quality" clauses and maintain higher "quality" clauses. However, the definition for quality is not entirely precise. Clause length is an intuitive proxy for quality where a shorter clause is considered better quality. The rationale is that a clause with n literals requires $n-1$ literals being assigned to false before the clause induces propagation so a smaller n means the clause is more likely to cause propagation. Audemard and Simon noticed that many of these $n-1$ literals are assigned together. More precisely, literals with the same decision level, called a *block*, are often logically connected in a way that assigning one propagates the others. They used this observation to devise a better clause quality metric called the *literal blocks distance* (LBD) [10] of a clause defined as the number of distinct decision levels of all the literals in that clause. A low LBD is intuitively better for the same reason as clause length, in that it takes fewer assignments before the clause propagates. Modern clause deletion heuristics delete learnt clauses with high LBD since these clauses are deemed to be lower quality.

Ultimately, a CDCL SAT solver is trying to automatize general resolution. For an unsatisfiable instance, there exists a shortest sequence of resolution rule applications that derives the empty clause, and hence a proof of unsatisfiability. If the sequence is short, then a CDCL SAT solver can solve this instance quickly given that CDCL p-simulates general resolution [75]. However this p-simulation result is contingent on nondeterministic branching which is infeasible to implement in practice. If we view CDCL as a resolution engine, then the branching heuristic design problem can be thought of as designing a deterministic branching heuristic that minimizes the number of resolution steps to the empty clause. Alternatively, this can be thought of as approximating the ideal nondeterministic branching as closely as possible.

Modern branching heuristics observe conflict analysis to dynamically rank the variables, see Section 4.2.1 for more details. The SAT solver can be split up into two parts: the "student" that constructs partial assignments and "teacher" that critiques them. The student consists of the branching and propagation whereas the teacher consists of conflict analysis and backjumping. Then the solver becomes a two player game between the student and the teacher. The student attempts to extend the partial assignment into a satisfying assignment. Meanwhile the teacher provides feedback to the student in the form of learnt clauses to inform the student of mistakes and repair the partial assignment. The student and teacher take turns building up and tearing down the partial assignment until the student concludes either satisfiable or unsatisfiable. From this model, the student-teacher

Algorithm 1 Pseudocode for a basic CDCL SAT solver. ϕ is the input CNF.

```
1: function CDCL(φ)
2:     ν ← ∅                                          ▷ Start with an empty assignment.
3:     loop
4:         ν ← BCP(φ, ν)
5:         if ∃conflictingClause ∈ φ such that Falsified(conflictingClause, ν) then
6:             if CurrentDecisionLevel(ν) = 0 then
7:                 return UNSAT
8:             end if
9:             learntClause ← ConflictAnalysis(conflictingClause, ν)
10:            φ ← φ ∪ {learntClause}                                  ▷ Clause learning
11:            ν ← BacktrackToDecisionLevel(DetermineBacktrackLevel(learntClause), ν)
12:        else if |ν| = |Variables(φ)| then
13:            return SAT
14:        else if RestartConditionMet() then
15:            ν ← BacktrackToDecisionLevel(0, ν)                      ▷ Restart
16:        else
17:            OptionallyPerformClauseDeletion()
18:            var ← BranchingHeuristic()
19:            value ← PolarityHeuristic(var)
20:            ν ← ν ∪ {var ↦ value}                                   ▷ Branch
21:        end if
22:    end loop
23: end function
```

feedback loop closely resembles the interplay between agent-environment loop in reinforcement learning literature. This similarity inspired us to pursue reinforcement learning as the basis for a new branching heuristic in Chapter 3.

2.1.3 Experimental Evaluation

SAT solvers are experimentally evaluated whenever a change is made, whether it be something small like a change of parameter or something big like implementing a new heuristic. The evaluations performed in this thesis adhere to the following.

Benchmark: SAT solvers are evaluated over a fixed set of CNF instances, called the benchmark. Hand selecting the benchmark can bias the evaluation since it is well known that certain solvers and/or heuristics perform better and worse on certain benchmarks. To avoid this issue of biasing the benchmark, we use the benchmarks from the SAT Competition which is standard practice in this field. The SAT Competition [1] is an annual competition where top SAT researchers worldwide compete for the fastest SAT solver. The benchmarks change every year. The instances come from diverse sources and require the solver to be well rounded to perform well. The competition typically splits the benchmarks into three categories. The application category are instances that come from real world usage of SAT solvers such as software and hardware verification. The hard combinatorial category are contrived instances designed to be challenging for modern solvers such as factoring and sub-graph isomorphism. The random category are instances that are randomly generated. We typically evaluate on the application and hard combinatorial categories since CDCL SAT solvers are extremely poor at solving random instances.

Environment: Experiments are performed in the cloud due to the size of the benchmarks. Cloud environments with heterogeneous hardware are unqualified for evaluating solvers with respect to solving time. All time-sensitive experiments were conducted on StarExec [82], a platform purposefully designed for evaluating SAT solvers. The machines on StarExec have the following specifications.

- Intel Xeon CPU E5-2609 at 2.40GHz
- 10240 KB cache
- 129022 MB main memory
- Red Hat Enterprise Linux Server release 7.2 (Maipo)

- Linux kernel 3.10.0-514.16.1.el7.x86_64
- gcc-4.8.5-4.el7.x86_64
- glibc-2.17-157.el7_3.1.i686

For time-sensitive experiments, we sometimes opt for the Orca cluster on the SHARCNET cloud [2] since it provides orders of magnitude more compute resources than StarExec. The heterogeneous machines on the Orca cluster have the following specifications.

- 24 core AMD Opteron at 2.2 GHz with 32.0 GB of main memory
- 16 core Intel Xeon at 2.6 GHz with 32.0 GB of main memory
- 16 core Intel Xeon at 2.7 GHz with 64.0 GB of main memory
- 16 core Intel Xeon at 2.7 GHz with 128.0 GB of main memory

Plotting: Results of an experiment are typically plotted in a *cactus plot* which makes it easy to see high level changes to performance at a glimpse. To construct a cactus plot, a solver is run on every instance in the benchmark and the solving time for each instance is recorded. All instances that timeout are discarded. The cactus plot has two axes. The x-axis is the number of instances solved and the y-axis is the solving time. For every value of $x > 0$, let y be the solving time in seconds for the x^{th} fastest instance to solve. Then the point (x, y) is added to the cactus plot. This is interpreted as x instances in the benchmark have solving times less than or equal to y seconds for the given solver. Being further to the right means solving more instances. Being further down means solving instances faster. See Figure 2.1 for an example of a cactus plot.

2.2 Average

A time series $t = \langle t_1, t_2, ..., t_n \rangle$ is a sequence of numbers where $t_i \in \mathbb{R}$ represents a data point collected at time i. For example, t_i can represent the distance traveled by a runner after i seconds. We can compute the average of a time series to express what a typical value is in the sequence. Two common methods for averaging includes the mean and the exponential moving average.

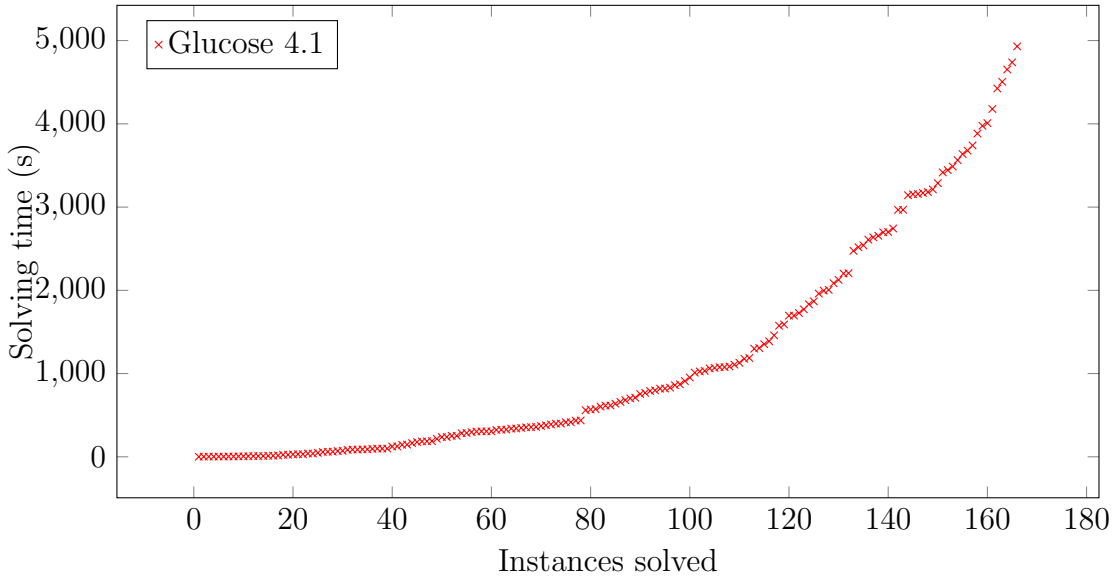

Figure 2.1: A cactus plot of the Glucose 4.1 solver over the 350 instances from the main track of SAT Competition 2017 with a 5000 second timeout.

2.2.1 Mean and Variance

The mean μ is a weighted sum of the time series where the weight is the constant $\frac{1}{n}$.

$$\mu(t) = \sum_{i=1}^{n} \frac{1}{n} t_i \tag{2.5}$$

Since every element in the time series is given equal weight, the temporal feature of the data is essentially ignored and the order of the time series does not matter. Also note that the expected value of the time series is also the mean.

$$\mathbf{E}(t) = \mu(t) \tag{2.6}$$

The variance σ^2 measures how far the data spreads from the mean. A low variance means most of the data is clustered around the mean. A high variance means the data is very spread out. The variance of a time series is defined as follows.

$$\sigma^2(t) = \frac{1}{n-1}\sum_{i=1}^{n}(t_i - \mu(t))^2 \tag{2.7}$$

The mean μ and variance σ^2 can be computed incrementally using the following formula.

$$\begin{aligned}
\delta_n &= t_n - \mu(\langle t_1, t_2, ..., t_{n-1}\rangle) && (2.8)\\
\mu(\langle t_1, t_2, ..., t_n\rangle) &= \mu(\langle t_1, t_2, ..., t_{n-1}\rangle) + \delta_n/n && (2.9)\\
\mu(\langle\rangle) &= 0 && (2.10)\\
\Delta_n &= t_n - \mu(\langle t_1, t_2, ..., t_n\rangle) && (2.11)\\
M(\langle t_1, t_2, ..., t_n\rangle) &= M(\langle t_1, t_2, ..., t_{n-1}\rangle) + \delta_n\Delta_n && (2.12)\\
M(\langle\rangle) &= 0 && (2.13)\\
\sigma^2(\langle t_1, t_2, ..., t_n\rangle) &= \frac{M(\langle t_1, t_2, ..., t_n\rangle)}{n-1} && (2.14)
\end{aligned}$$

2.2.2 Exponential Moving Average (EMA)

Time is an important aspect when analyzing data, as old data can distort analysis. For example, consider a time series t_i representing the points scored by an athlete in his i^{th} game. One might compute the mean of the time series to evaluate the quality of the player in his current form to determine whether he is worth adding to the team. However, the mean would give a misleading picture of his current form since his first game he played as a child and his most recent game are weighted equally. What makes more sense is to give more weight to his recent games since this is more reflective of his current form. The exponential moving average (EMA) does exactly that by giving a weighted average with more recent elements having exponentially higher weights [22]. Typically, EMA is defined recursively with the parameter $0 < \alpha < 1$ called *smoothing factor* or *step-size* depending on context.

$$\begin{aligned}
EMA_\alpha(\langle t_1, t_2, ..., t_n\rangle) &= \alpha \cdot t_n + (1-\alpha)\cdot EMA_\alpha(\langle t_1, t_2, ..., t_{n-1}\rangle) && (2.15)\\
EMA_\alpha(\langle\rangle) &= 0 && (2.16)
\end{aligned}$$

By convention, we will define the EMA of an empty time series to be zero, although in practice this may not be ideal. As we will see in the branching heuristic case, setting

the value for the EMA of an empty time series corresponds to the initialization problem of branching heuristics.

The above recursive definition gives a simple incremental implementation as more data comes in. If Q is the EMA of $\langle t_1, t_2, ..., t_n \rangle$ and a new data point t_{n+1} is collected, then $Q := \alpha \cdot t_{n+1} + (1 - \alpha) \cdot Q$ will be the EMA of the new time series containing t_{n+1}. Note that the time series does not need to be stored to compute the EMA, only Q needs to be stored. This makes the EMA computation very cheap in both time and space.

In any case, the above recursive definition can be expanded to a closed form formula.

$$EMA_\alpha(\langle t_1, t_2, ..., t_n \rangle) = \alpha \sum_{i=1}^{n} (1 - \alpha)^{n-i} t_i \tag{2.17}$$

In this expanded form, it is clear that EMA is also a weighted average where t_i is given the weight $\alpha(1 - \alpha)^{n-i}$. The weights shrink exponentially as i decreases. Based on the coefficient $(1 - \alpha)^{n-i}$, it is clear that a smoothing factor α closer to 1 discounts older data more and hence puts more weight in the recent data of the time series. Finding a good value for α is a mix of intuition and experimentation.

2.3 Probability Distribution

A *probability distribution* describes the probability of all possible outcomes of a random variable X. There are two classes of probability distributions, the discrete probability distribution and the continuous probability distribution. In this section, we describe the latter.

The continuous probability distribution, as the name suggests, is applicable when the outcomes are continuous such as real numbers. It associates probabilities to outcomes via a *cumulative distribution function* $F_X(x)$ for a random variable X.

$$F_X(x) = P(X \leq x) \tag{2.18}$$

The *probability density function* $f_X(x)$ is defined with respect to the cumulative distribution function that describes the relative likelihood of the outcomes in the continuous domain.

$$f_X(x) = \frac{d}{dx} F_X(x) \tag{2.19}$$

2.3.1 Mean and Variance

Suppose we sample n times the random variable X resulting in $\langle t_1, t_2, ..., t_n \rangle$. The *mean* of X is the mean of the time series $\mu(\langle t_1, t_2, ..., t_n \rangle)$ as n approaches infinity, also known as the law of large numbers. Alternatively, the mean of X can be computed using the probability density function.

$$\mu(X) = \int x f_X(x) dx \tag{2.20}$$

The variance is defined as the following.

$$\sigma^2(X) = \int (x - \mu(X))^2 f_X(x) dx \tag{2.21}$$

2.3.2 Normal Distribution

The *normal distribution* is a common continuous probability distribution defined by two parameters: the mean μ and variance σ^2. The variance is sometimes replaced by the standard deviation σ, the square root of the variance. Even though the normal distribution is a continuous distribution, it is often used to approximate discrete distributions. The probability density function $f(x)$ for the normal distribution is defined as follows.

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{2.22}$$

The probability density function forms a bell curve around the mean μ. The width of the bell curve is proportional to the standard deviation σ. In this thesis, we use the normal distribution to carve out percentiles, that is, we use it compute $P(X \leq x)$ where $X \sim norm(\mu, \sigma^2)$. This can be calculated using z-score defined as follows.

$$z = \frac{x - \mu}{\sigma} \tag{2.23}$$

Once z is computed, use a Z-table to compute $\Phi(z)$. The Z-table is a table displaying the values of Φ for common values of z. The probability can then be computed as $P(X \leq x) = \Phi(z) = \Phi(\frac{x-\mu}{\sigma})$.

The reverse is also possible, find x such that $P(X \leq x) = p$ where p is a constant. In other words, find the value x with percentile p.

$$P(X \leq x) = p \tag{2.24}$$
$$\Phi(\frac{x-\mu}{\sigma}) = p \tag{2.25}$$
$$\frac{x-\mu}{\sigma} = \Phi^{-1}(p) \tag{2.26}$$
$$x = \Phi^{-1}(p)\sigma + \mu \tag{2.27}$$

Given that $\Phi^{-1}(p)$ is a constant for a fixed p, x can be computed by knowing the mean μ and standard deviation σ.

2.4 Machine Learning

Machine learning is a rich field in artificial intelligence broadly to learn from data. We will focus on two paradigms: supervised learning and reinforcement learning.

2.4.1 Supervised Learning

Suppose there exists some complex function $f : Input \rightarrow Output$ that is infeasible to implement by hand. 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*, machine learning algorithms infer a new function $\tilde{f}$ that approximates f by analyzing the training set. These types of machine learning algorithms are called *supervised learning* algorithms.

For example, consider the problem of image recognition: given an image as input, return as output the name of the object present in the image. In this case the function f maps inputs to names $f : Image \rightarrow Name$. It is easy for our human brains to execute f, given an image we can instantly recognize the object in it. However, implementing f as a computer program by hand is infeasible. But engineers are interested in f for image recognition to apply it in autonomous driving for example. In practice, these engineers use supervised learning to approximate f. One of the challenges is to collect enough data to train a complex function like f. For image recognition, the labeled training data is of type $\langle Image, Name\rangle$ consisting of an image along with the correct name of the object present

in the image. Humans can manually provide the labels, the names of the objects in the images in this case. Once the training set is acquired, the supervised learning algorithm learns $\tilde{f}$.

If everything goes well, $\tilde{f}$ returns 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, $\tilde{f}$ returns 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 $\tilde{f} : \mathbb{R}^n \to Output$ 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. For example, in image recognition, the feature extraction function $Image \to \mathbb{R}^n$ transforms an image into an array of pixel RGB values where n is the number of pixels in the image.

Supervised learning is divided into two categories depending on the domain of the output. *Regression* is the subclass of supervised learning where the output is continuous. *Classification* is the other subclass where the output is categorical.

Regression: In this thesis, we only consider *linear regression* where $\tilde{f} : \mathbb{R}^n \to \mathbb{R}$ is a linear function.

$$\tilde{f}([x_1, x_2, ..., x_n]) := w_0 + w_1 x_1 + w_2 x_2 + ... + w_n x_n \tag{2.28}$$

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.

Classification: In this thesis, we only consider binary classification where the function we want to learn has the type $f : Input \to \{1, 0\}$. In other words, f maps every input to either the class 1 or the class 0. f is also called a predicate.

We use *logistic regression* [30], a popular technique for binary classification, to learn a function $\tilde{f}$ that cheaply approximates f. Despite the word "regression" in logistic regression, it is typically used for classification. The function learnt by logistic regression has the type $\tilde{f} : \mathbb{R}^n \to [0, 1]$ 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. Equivalently, this is the

probability that the output is not in class 0. Logistic regression defines the function $\tilde{f}$ as follows.

$$\tilde{f}([x_1, x_2, ..., x_n]) := \sigma(w_0 + w_1x_1 + w_2x_2 + ... + w_nx_n), \quad \sigma(z) := \frac{1}{1 + e^{-z}} \tag{2.29}$$

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. Note that $w_0 + w_1x_1 + w_2x_2 + ... + w_nx_n$ is a linear function like in linear regression. 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 interpreted as a probability.

The learning algorithm we use to set the weights for both linear regression and logistic regression is called *stochastic gradient descent* (SGD) [20], which is a popular algorithm in many supervised learning problems. SGD minimizes the error by taking a step in the opposite direction of the gradient with respect to each data point. The error of a data point can be computed by the following error function, which is defined differently for linear regression and logistic regression.

Linear regression:

$$Err(\mathbf{x}, y, \mathbf{W}) = (y - \tilde{f}(\mathbf{x}; \mathbf{W}))^2 \tag{2.30}$$

$\mathbf{x}$ is the input of a labeled training data point, y is the corresponding label for this data point, and $\mathbf{W}$ is a vector weights that parameterizes $\tilde{f}$.

Logistic regression:

$$Err(\mathbf{x}, y, \mathbf{W}) = y(1 - \tilde{f}(\mathbf{x}; \mathbf{W})) + (1 - y)(\tilde{f}(\mathbf{x}; \mathbf{W})) \tag{2.31}$$

$\mathbf{x}$ is the input of a labeled training data point, y is the corresponding target class (0 or 1) for this data point and $\mathbf{W}$ is a vector weights that parameterizes $\tilde{f}$.

In either case, the error function intuitively measures the difference between the output of $\tilde{f}$ and the actual labeled output in the training set. If the error is high, then the fit is poor. The duty of the learning algorithm is to select the weight vector $\mathbf{W}$ such that the error function is minimized. For linear and logistic regression, this boils down to a convex

optimization problem where the objective is to minimize the error function by searching over the space of weights. The local minimum and global minimum are one and the same for convex optimization. The SGD algorithm which finds local minimums are ideal for linear regression and logistic regression.

SGD minimizes the error rate by computing the slope or *gradient* of the error function with respect to the weights, and moves the weights in the direction where the slope points downwards hence reducing the error. SGD takes a step in the opposite direction of the gradient as follows.

$$\mathbf{W}' \leftarrow \mathbf{W} - \alpha \frac{\partial Err(\mathbf{x}, y, \mathbf{W})}{\partial \mathbf{W}} \tag{2.32}$$

Here α is the step length (also known as the learning rate, not to be confused with the unrelated definition of learning rate in SAT solving context). 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 [69] to mitigate overfitting. L2 regularization introduces a new term in the error function that favors small weights.

Linear regression:

$$Err(\mathbf{x}, y, \mathbf{W}) = (y - \tilde{f}(\mathbf{x}; \mathbf{W}))^2 + \lambda ||\mathbf{W}||_2^2 \tag{2.33}$$

Logistic regression:

$$Err(\mathbf{x}, y, \mathbf{W}) = y(1 - \tilde{f}(\mathbf{x}; \mathbf{W})) + (1 - y)(\tilde{f}(\mathbf{x}; \mathbf{W})) + \lambda ||\mathbf{W}||_2^2 \tag{2.34}$$

Essentially the error function appends the term $\lambda ||\mathbf{W}||_2^2$ at the end to perform L2 regularization where λ is a constant.

$$||\mathbf{W}||_2^2 := \sum_{i=0}^{n} w_i^2 \tag{2.35}$$

Recall that SGD minimizes the error function. By adding term $||\mathbf{W}||_2^2$, the error grows if the weights w_i are large. So SGD must also keep the weights small to minimize the

new error function. λ is a parameter that determines the importance of the regularization penalty. How regularization prevents overfitting is beyond the scope of this thesis.

SGD is also commonly used in an online and incremental 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 for designing heuristics.

2.4.2 Reinforcement Learning

In psychology, reinforcement refers to increasing or decreasing the likelihood of certain actions based on positive (i.e., reward) or negative (i.e., cost) stimulus. For example, rewarding a dog for good behavior teaches the dog to behave properly. Likewise, punishing a dog for poor behavior teaches the dog to cease its bad behavior. Inspired by reinforcement in psychology, an area of machine learning called *reinforcement learning* models an agent who learns which actions are good or bad based on feedback it receives from performing those actions.

Reinforcement learning is a very deep field. In this thesis, we will focus on the simplest formulation of reinforcement learning called *multi-armed bandits* or MAB for short. We will explain the MAB problem [83] through a classical analogy of a gambler (also called *agent*) in a casino with n slot machines. The objective of the gambler is to maximize the *reward* of monetary payouts received by playing these slot machines. Each slot machine has a *reward probability distribution* describing its monetary 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 has n *actions* to choose from corresponding to playing one of the n slot machines. The gambler picks an action, plays the corresponding slot machine, and receives a reward in terms of monetary payout by sampling that slot machine's reward probability distribution. The MAB problem is to decide which actions to take to maximize the cumulative reward over the long term, that is, to make as much money as possible. Next, we describe a few algorithms that solve the MAB problem.

Optimal: If the reward probability distributions of the slot machines were revealed, then the gambler would simply play the slot machine whose reward probability distribution has the highest mean. This will maximize expected monetary payouts for the

gambler. Note that this optimal strategy only requires knowing the mean of the reward probability distribution for each slot machine and nothing else. The following two algorithms approximate this optimal strategy by estimating the mean reward. This is denoted by Q_a where a is the action and Q_a is the estimated mean reward for playing action a.

Sample-Average: Since the reward probability distribution is hidden, a simple MAB algorithm called *sample-average* [83] estimates the true mean of each reward probability distribution by the sample mean of the rewards observed from playing the slot machines. If t is the time series of rewards received for playing action a, then $Q_a = \mu(t)$. When it comes time to select an action, the sample-average algorithm picks $argmax_a Q_a$, that is, the machine it estimates to have the highest mean reward.

For example, suppose there are 2 slot machines named A and B. The gambler plays both slot machines 4 times each, receiving the 4 rewards $\langle \$1, \$2, \$3, \$4 \rangle$ for A and $\langle \$5, \$4, \$3, \$2 \rangle$ for B. Then the algorithm will estimate the mean rewards of slot machines A and B as $Q_A = \mu(\langle \$1, \$2, \$3, \$4 \rangle) = \$2.5$ and $Q_B = \mu(\langle \$5, \$4, \$3, \$2 \rangle) = \$3.5$ respectively. Since $Q_B > Q_A$, the sample-average algorithm opts to play slot machine B.

By the law of large numbers from probability theory, the estimated mean reward will converge to the true mean reward as the number of observations approaches infinity. In other words, the gambler improves the accuracy of its estimate of mean reward for a slot machine by simply playing that slot machine to generate more reward data. More data means better estimates.

Exponential Recency Weighted Average: The sample-average algorithm is applicable if the hidden reward 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 (respectively bigger and bigger) rewards the more it has been played. The older the observed reward, the bigger the difference between the current probability distribution and the distribution from which the reward was sampled. Hence, older observed rewards should have a smaller impact on the estimated mean reward than recent observed rewards. This gives rise to the *exponential recency weighted average* [83] (ERWA) algorithm. Instead of computing the mean of the observed rewards as in the sample-average algorithm, ERWA uses EMA to give higher weights to recent observations relative to distant observations. It estimates the mean reward as $Q_a = EMA(t)$ where t is the time series of rewards received for playing action a. Using the same example as the sample-average, ERWA estimates the mean payout of the slot machines A and B as

$Q_A = EMA_\alpha(\langle \$1, \$2, \$3, \$4 \rangle) = \$3.0625$ and $Q_B EMA_\alpha(\langle \$5, \$4, \$3, \$2 \rangle) = \$2.5625$ respectively where the smoothing factor is $\alpha = 0.5$. Since $Q_A > Q_B$, ERWA estimates the slot machine A to have a higher mean and opts to play slot machine A. This differs from the sample-average algorithm which selected the slot machine B. The recent rewards for the slot machine A is higher than the recent rewards of the slot machine B, and this factored into ERWA's decision to rank slot machine A as the better machine to play.

In the sample-average and ERWA algorithms, selecting the action with the highest estimated mean reward is called *exploitation*, that is, it chose the action it believes at that time to be the best action given extant observations. The word exploitation refers to exploiting the current knowledge accrued from playing the various slot machines and observing their rewards in the past. However, this strategy can be suboptimal if the estimates of the mean rewards are inaccurate.

For example, suppose a casino has 2 slot machines, with the first and second machines having reward probability distributions with means of 100 and 200 respectively. The gambler plays the first slot machine ten times and estimates its mean reward to be 105. The gambler then plays the second machine ten times and by sheer bad luck receives uncharacteristically low rewards. After the ten plays, the gambler estimates the mean reward to be 50 for the second slot machine. If the gambler were to purely exploit, he would continue playing the first slot machine perhaps perpetually. As long as the estimated mean reward of the first slot machine remains above 50, the gambler never plays the second machine. The estimated reward of the second slot machine remains at 50, and the gambler never learns the true mean of 200. So by only exploiting, the gambler can fail to learn that the second slot machine is the optimal slot machine to play. To avoid this problem, the gambler needs to play the second machine more often, even though at the current time it estimates the first slot machine to be the optimal slot machine to play. That way it has the opportunity to collect more data about the second slot machine to produce a more accurate estimate of its mean reward.

To address the shortcomings of exploitation, some algorithms introduce an aspect called *exploration* [83] where it plays an action whose estimated mean reward is not the highest. The word exploration refers to exploring other actions to gather more data about them to improve the accuracy of the estimate mean. However, too much exploration is also a problem. With enough data, the gambler has a reasonable picture of which slot machine has the highest mean reward. Exploring in this case means playing a suboptimal slot machine. This leads to the *exploitation versus exploration* trade-off where too much exploitation

leads to the gambler learning the optimal action much later and too much exploration leads to the gambler selecting suboptimal actions too often.

2.5 Graph Community Structure and Centrality

Variables in a CNF formula have logical connections and the relationships between variables is important to heuristics. The CNF can be represented graphically to visualize these relationships. Once the CNF is in the form of a graph, we can apply various graph metrics to analyze the CNF. We primarily focus on *community structure* and *centrality* metrics to identify vertices of interest in the graph and correlate these with the variables prioritized by VSIDS in Section 4.2.3. Community structure groups vertices that are closely related. Centrality is a class of metrics to measure the "importance" of various vertices in a graph by assigning a numerical value to each vertex. The number associated with each vertex denotes its relative importance in the graph [36, 34].

2.5.1 Variable Incidence Graph

Variables in the same clause can propagate each other. The *variable incidence graph* (VIG) captures these forms of relations between variables. In the VIG, every variable is its own vertex. Two vertices share an edge if the corresponding variables share a clause. Since the VIG captures potential propagations between variables, and shorter clauses are more likely to propagate, naturally the VIG assigns higher weights to edges corresponding to shorter clauses. If two variables share a clause of length $|C|$, then the weight of the edge between them is assigned to $\frac{1}{|C|-1}$. This weighting scheme was chosen so that every clause adds a weighted degree of 1 to all variables in the clause. All edges between each pair of vertices is combined into a single weighted edge by summing the weights.

Definition 3 (Variable incidence graph (VIG)) *Let ϕ be a CNF formula. The variable incidence graph is a defined by the set of vertices V , the set of edges E, and the weights of edges $W : E \to \mathbb{R}$.*

$$V = Variables(\phi) \tag{2.36}$$

$$E = \{xy \mid C \in \phi \wedge x \in C \wedge y \in C \wedge x \neq y\} \tag{2.37}$$

$$W(xy) = \sum_{C \in \phi \wedge x \in C \wedge y \in C \wedge x \neq y} \frac{1}{|C| - 1} \tag{2.38}$$

When appropriate, we sometimes interpret a VIG as an *adjacency matrix* when convenient. The adjacency matrix is a matrix representation of the weight function.

Definition 4 (Adjacency matrix) *Suppose all the vertices are arbitrarily numbered as* $x_1, x_2, ..., x_n$. *An adjacency matrix* A *is a* $n \times n$ *matrix, where* n *is the number of variables and the cell* $A_{ij} = W(x_i x_j)$.

2.5.2 Community Structure

The concept of decomposing graphs arose in the study of complex networks [39, 28] such as social networks. A graph has *community structure* if the graph can be partitioned into subgroup of vertices called *communities* subject to the following.

1. Each community has many edges between vertices within the same community. We refer to these edges as *intracommunity* edges.
2. For every pair of distinct community, there are very few edges connecting vertices between these two communities. We refer to these edges as *bridges*.

Given a partition of variables, the quality of the community structure for this particular partition is measured by the *modularity* denoted by Q[28].

$$Q = \sum_i (e_{ii} - a_i^2) \tag{2.39}$$

$$e_{ij} = \frac{1}{2m} \sum_v \sum_w \delta(c_v, i)\delta(c_w, j) \tag{2.40}$$

$$a_i = \frac{1}{2m} \sum_v k_v \delta(c_v, i) \tag{2.41}$$

$$k_i = \sum_j A_{ij} \tag{2.42}$$

$$m = \frac{1}{2} \sum_i k_i \tag{2.43}$$

$$\delta(i, j) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \tag{2.44}$$

c_v is the community in which v belongs to according to the partition. k_i is the weighted degree of vertex i. m is the total weight of all edges in the graph (the $\frac{1}{2}$ is to prevent double counting the weights of an undirected graph). e_{ij} is the weighted fraction of bridge edges that start in community i and end in community j. a_i is the weighted fraction of edges incident to vertices in community i. From the definition of network modularity $Q = \sum_i (e_{ii} - a_i^2)$, it is clear that the term e_{ii} boosts Q for intracommunity edges whereas a_i penalizes Q for bridges. Q ranges between 0 and 1. We say that a graph has a "good" community structure if there exists a partition of communities such that Q is close to 1.

The ideal community structure detection algorithm finds the partition of variables into communities such that Q is maximized. However, this exact maximization is prohibitively expensive. In practice, there are a variety of algorithms that maximizes Q on a best effort basis and tends to work well for most use cases. We refer the reader to these papers [39, 28, 89] for a more formal introduction to community structure of graphs.

Recently there has been some interesting discoveries showing that SAT formulas from real world applications exhibit community structure [7]. Subsequently, the authors of the paper [72] showed that the running time of CDCL solvers is correlated with community structures of SAT instances.

In the context of the community structure of the VIG of a CNF, *bridge variables* are the variables corresponding to bridges in the community structure.

Definition 5 (Bridge Variable) *A bridge variable is a variable whose corresponding vertex in the VIG is incident to a bridge edge.*

2.5.3 Degree Centrality

Degree centrality is motivated by social networks where vertices are people, and edges represents that two people are acquaintances. The person with the most acquaintances is the most popular. Degree centrality measures popularity. The degree centrality [34] of a vertex in a graph is defined as the degree of the vertex. More precisely, degree centrality of a vertex is computed as follows.

$$Degree(x) = \sum_{y \in V} W(xy) \tag{2.45}$$

2.5.4 Eigenvector Centrality

Degree centrality only takes into account the weight of the edges. Eigenvector centrality also accounts for the importance of the vertex on the opposite side of the edge. For example, being an acquaintance with the president is much more important than being an acquaintance with an aide. The eigenvector centrality [71] is defined as follows.

$$Eigen(x) = \lambda^{-1} \sum_{y \in V} W(xy) Eigen(y) \tag{2.46}$$

λ is a constant corresponding to the largest eigenvalue. Note that the definition is recursive, making it less straightforward to compute than degree centrality. We use the power iteration [41] method to approximate the eigenvector centrality.

To execute the power iteration method, the graph is first represented as an adjacency matrix A. Let $\mathbf{b_0}$ be a vector of real numbers used as an initial estimation of the eigenvector centrality. The i^{th} element of the vector $\mathbf{b_0}$ is the estimate of the eigenvector centrality of i_{th} variable. We use the vector of all $\frac{1}{\sqrt{n}}$ as our initial estimation. An iteration of the power iteration method updates the estimates like so.

$$\mathbf{b}_{k+1} = \frac{A\mathbf{b}_k}{||A\mathbf{b}_k||_2^2} \qquad \text{(Power iteration)}$$

$||M||_2^2$ of a matrix M is computed as follows.

$$||M||_2^2 = \sum_i \sum_j m_{ij}^2 \tag{2.47}$$

The power iteration method converges the vector b_k to the eigenvector centrality.

$$\lim_{k \to \infty} Eigen = \mathbf{b}_k \tag{2.48}$$

The more iterations of power iteration, the closer b_k approximates the eigenvector centrality. Since this computation is rather expensive for large graphs, we use only 100 iterations. For our use case, the estimated centrality values change very little after 100 iterations.

Chapter 3

Machine Learning-Based Branching Heuristics

In this chapter, we design new state-of-the-art branching heuristics by exploiting machine learning. One of the challenges in designing branching heuristics is that it is not clear what constitutes a good decision variable. Ideally, the branching heuristic selects the decision variables such that the total solving time is minimized, but implementing this ideal branching heuristic is impractical since the solving time is not known until the solver terminates. Despite this setback, we still need to classify good decision variables in designing branching heuristics. In this chapter, we define a simple metric called *global learning rate (GLR)* that negatively correlates with solving time. In lieu of directly minimizing solving time, we opt to maximize GLR instead for practical reasons. With this perspective, good branching variables are those that increase GLR if branched on. Viewing the branching problem explicitly as a GLR maximization problem gives clarity and structure to designing branching heuristics by SAT researchers. This chapter proves that this perspective of branching is very profitable.

Designers of branching heuristics must balance a trade-off between selecting good decision variables and the time it takes to compute these decision variables. Many instances, especially those originating from cryptography or combinatorial mathematics, requires brute force to exhaust certain regions of the search space. A costly branching heuristic performs poorly in these brute force situations. On the other hand, we know from work on backdoors [86] that selecting the correct decision variables can result in very small search spaces in practice. What we find is that machine learning is a powerful class of algorithms that manages this trade-off by cheaply learning the good decision variables. In this chap-

ter, we define our objective, the global learning rate (GLR), and present some machine learning-based branching heuristics to maximize the GLR objective.

3.1 Global Learning Rate (GLR)

The branching heuristic does not affect completeness nor soundness of CDCL, it is simply there to improve performance. Ultimately the branching heuristic is solving an optimization problem, select the branching order such that solving time is minimized. Hence the problem of branching is an optimization problem where the solving time is the objective to minimize. However, it is infeasible to calculate the solving time a priori, which makes it unsuitable as an objective to minimize for branching.

The goal is to define a metric of CDCL solvers that serves as a proxy for solving time. In other words, this feature empirically separates good and bad branching heuristics. This serves two purposes. First, this helps us formalize an intuition on what makes a branching heuristic good. Second, this feature can be used as an objective in an optimization problem as a substitute for solving time. 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. Hence learning new clauses gives a simple metric for progress of the search.

We propose the *global learning rate* (GLR) as a proxy for solving time of a solver. GLR is defined as follows.

$$GLR := \frac{\#\ of\ conflicts}{\#\ of\ decisions} \qquad \text{(Global Learning Rate)}$$

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

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. First 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 [52], 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 1-UIP [66] since it is universally used by all modern CDCL solvers.

We propose the following hypothesis regarding GLR. We support this hypothesis in a series of experiments.

Hypothesis 1 *The branching heuristic that achieves higher GLR tends to be a better branching heuristic.*

3.2 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 compete with practical branching heuristics, it provides a proof of concept for GLR maximization and a gold standard for subsequent branching heuristics to emulate.

We define the function $c : PA \rightarrow \{1, 0\}$ 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 encounters 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.

$$c(PA) = \begin{cases} 1 & \text{if } BCP(ClauseDatabase, PA) \text{ results in a conflict} \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

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 polarity) 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 2 for the implementation of GGB.

Algorithm 2 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 pseudocode) and makes it greedier by causing conflicts whenever possible. In general, VSIDS can be replaced with any other branching heuristic.

```
1: function PhaseSaving(Var)                    ▷ Return the variable plus a polarity.
2:     return mkLit(Var, Var_savedPolarity)
3: end function
4:
5: function VSIDS(Vars)          ▷ Return variable with highest VSIDS activity plus a
   polarity.
6:     return PhaseSaving(argmax_{v∈Vars} v_activity)
7: end function
8:
9: function GGB
10:    CPA ← CurrentPartialAssignment
11:    V ← UnassignedVariables
12:    oneClass ← {v ∈ V  |  c(CPA ∪ {PhaseSaving(v)}) = 1}
13:    zeroClass ← V \ oneClass
14:    if oneClass ≠ ∅ then                     ▷ Next BCP will cause a conflict.
15:        return VSIDS(oneClass)
16:    else                                     ▷ Next BCP will not cause a conflict.
17:        return VSIDS(zeroClass)
18:    end if
19: end function
```

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 factored out, then GGB would be a more efficient heuristic than 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.

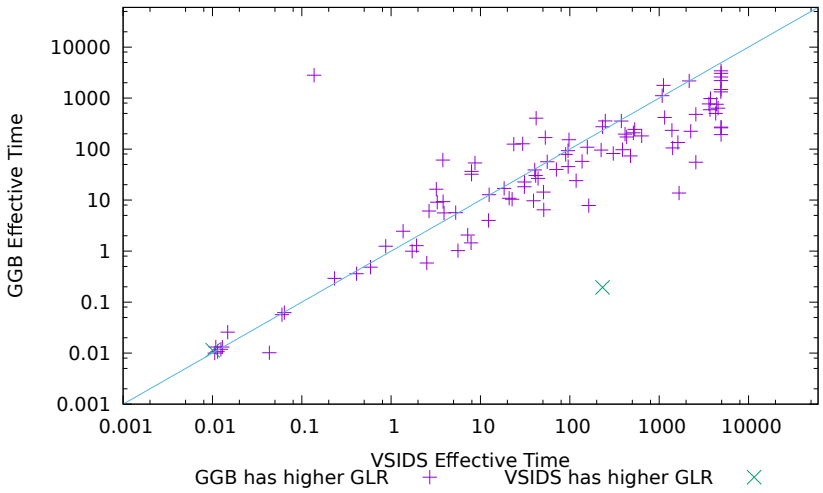

Figure 3.1: GGB vs VSIDS. Each point in the plot is a comparable instance. Note that the axes are in log scale. GGB has a higher GLR for all but 2 instances. GGB has a mean GLR of 0.74 for this benchmark whereas VSIDS has a mean GLR of 0.59.

3.2.1 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 [82], a platform purposefully designed for evaluating SAT solvers. Restarts and clause deletion were 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 3.1 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

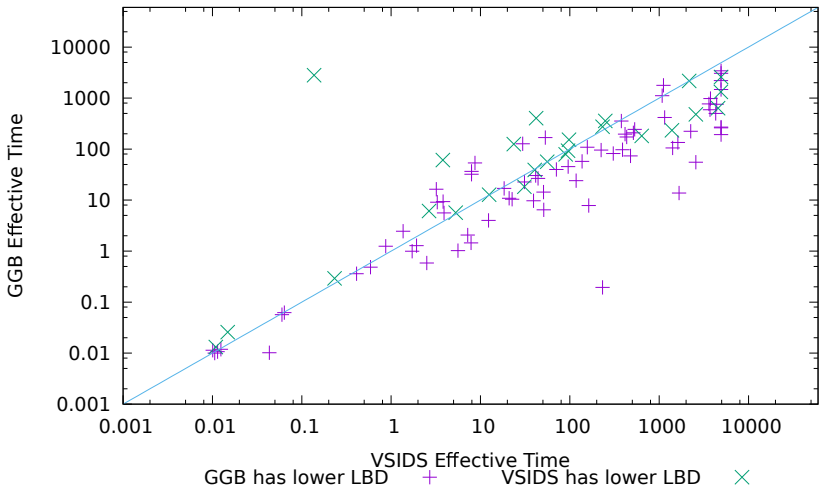

Figure 3.2: GGB vs VSIDS. GGB has a lower average LBD for 72 of the 98 comparable instances. GGB has a mean average LBD of 37.2 for this benchmark whereas VSIDS has a mean average LBD of 61.1.

as expected. Figure 3.2 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 earlier when the decision level is low, which keeps the LBD small since LBD cannot exceed the current decision level.

3.3 Multi-Armed Bandits Branching

As noted earlier, the student-teacher model of CDCL portrays a feedback loop between branching and clause learning that is highly reminiscent of reinforcement learning. We pursue reinforcement learning in this section as a means for designing new branching heuristics. However, we need to modify the GLR objective slightly to work in the reinforcement learning framework.

3.3.1 Learning Rate Objective

Branching is an *action* in reinforcement learning parlance. To design a branching heuristic with reinforcement learning, we need to design a reward for individual actions that guide the solver towards higher GLR. Although the ultimate goal is to increase GLR, the GLR metric is a "global" metric that measures the overall performance. We define a local version of GLR, simply called *learning rate*, that measures the contribution to GLR by individual actions/variables.

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 $\frac{2}{5}$.

In other words, the learning rate of a variable is the probability the variable generates a conflict when assigned. Clearly branching on variables with high learning rate increases the global learning rate.

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. These circumstances are well-suited for multi-armed bandits.

3.3.2 Multi-Armed Bandit Model for Branching

Given n Boolean variables, we abstract branching 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* are 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* are used interchangeably. The MAB algorithm uses the reward to update its internal estimates of the action that maximizes 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 from 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 all receive their own individual rewards corresponding to their LR, and the MAB algorithm uses 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.

3.3.3 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.

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 is more efficient.

Algorithm 3 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 $Assigned_v$.
  end for
end procedure

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**
    $Participated_v \leftarrow Participated_v + 1$
  end for
  if $\alpha > 0.06$ **then**
    $\alpha \leftarrow \alpha - 10^{-6}$
  end if
end procedure

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

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.
  end if
end procedure

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.
end function

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 Section 3.3.1) 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 selects 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 3 for pseudocode of the implementation. Alternative to the above description, each variable v maintains a floating point number Q_v representing $EMA_\alpha(ts_v)$. Q_v is initialized to 0 at the start of the search. When v receives reward r, 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 30 of Algorithm 3). When the solver requests the next branching variable, the implementation selects the variable v^* where $v^* = argmax_{v \in U} Q_v$ and U is the set of currently unassigned variables (see line 36 of Algorithm 3). Note that Q_v can be stored in a priority queue for all unassigned variables v, hence finding the maximum takes 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 [83]. After each conflict, the step-size is decreased by 10^{-6} until it reaches 0.06 (see line 17 in Algorithm 3), and remains at 0.06 for the remainder of the run. This step-size management is equivalent to the one in CHB [59] and is similar to how the Glucose solver manages the VSIDS decay factor by increasing it over time [12].

Reason Side Rate (RSR) Extension: 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 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 25 in Algorithm 4). Note that we did not change the definition of the reward. The extension simply encourages ERWA to branch on variables with high RSR. We hypothesize that variables observed to have high RSR are likely to have high LR as well.

Locality Extension: Recent research shows that VSIDS exhibits locality [61], defined with respect to the community structure of the input CNF instance [61, 72, 7]. 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.

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 5). 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 [33, 88] 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 $Q_v \leftarrow 0.95^k \times Q_v$.

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.

3.3.4 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 without the extensions. Fourth, we show the

Algorithm 4 Pseudocode for ERWA as a branching heuristic with the RSR extension. The pseudocode $Algorithm1.method(...)$ is calling out to the code in Algorithm 3. The procedure $PickBranchLit$ is unchanged.

procedure INITIALIZE
  $Algorithm1.Initialize()$
  for $v \in Vars$ **do** ▷ $Vars$ is the set of Boolean variables in the input CNF.
    $Reasoned_v \leftarrow 0$ ▷ The number of learnt clauses v reasoned in generating since $Assigned_v$.
  end for
end procedure

procedure AFTERCONFLICTANALYSIS($learntClauseVars \subseteq Vars$, $conflictSide \subseteq Vars$)
  $Algorithm1.AfterConflictAnalysis(learntClauseVars, conflictSide)$
  for $v \in (\bigcup_{u \in learntClauseVars} reason(u)) \setminus learntClauseVars$ **do**
    $Reasoned_v \leftarrow Reasoned_v + 1$
  end for
end procedure

procedure ONASSIGN($v \in Vars$)
  $Algorithm1.OnAssign()$
  $Reasoned_v \leftarrow 0$
end procedure

procedure ONUNASSIGN($v \in Vars$)
  $Interval \leftarrow LearntCounter - Assigned_v$
  if $Interval > 0$ **then** ▷ $Interval = 0$ is possible due to restarts.
    $r \leftarrow Participated_v / Interval.$ ▷ r is the LR.
    $rsr \leftarrow Reasoned_v / Interval.$ ▷ rsr is the RSR.
    $Q_v = (1 - \alpha) \cdot Q_v + \alpha \cdot (r + rsr)$ ▷ Update the EMA incrementally.
  end if
end procedure

Algorithm 5 Pseudocode for ERWA as a branching heuristic with the locality extension. $AfterConflictAnalysis$ is the only procedure modified.

```
1: procedure AfterConflictAnalysis(learntClauseVars ⊆ Vars, conflictSide ⊆ Vars)
2:     Algorithm2.AfterConflictAnalysis(learntClauseVars, conflictSide)
3:     U ← {v ∈ Vars | isUnassigned(v)}
4:     for v ∈ U do
5:         Q_v ← 0.95 × Q_v.
6:     end for
7: end procedure
```

effectiveness of LRB within a state-of-the-art CDCL solver, namely, CryptoMiniSat [80]. To better gauge the results of these experiments, we quote two leading SAT solver developers, Professors Audemard and Simon [11]:

> "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."

Setup

The experiments are performed by running CDCL solvers with various branching heuristics on StarExec [82]. 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 7.5GB 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 [32] (simp version) with one modification to use the popular *aggressive LBD-based clause deletion* proposed by the authors of the Glucose solver in 2009 [10]. Since MiniSat is a relatively simple solver with very few features, it is ideal for our base solver to better isolate the effects of 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.

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 3 solvers over the entire benchmark and report the number of instances solved by these solvers within the time limit in Table 3.1. 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 Figure 3.3 for a cactus plot of the solving times.

Experiment: LRB vs VSIDS vs CHB

In this experiment, we compare LRB with the state-of-the-art branching heuristics VSIDS [68] and CHB [59]. 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 3.2. LRB solves a total of 1279 instances, VSIDS solves a total of 1179 instances, and CHB solves a total of 1235 instances. See Figure 3.3 for a cactus plot of the solving times.

Experiment: LRB and Learning Rate

In this experiment, we measure the efficacy of the 5 branching heuristics from Table 3.1 and Table 3.2 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 3.3. 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

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 (w/o duplicates)	**SAT**	**638**	**632**	**654**
	UNSAT	**574**	**619**	**625**
	BOTH	**1212**	**1251**	**1279**

Table 3.1: 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	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 (w/o duplicates)	**SAT**	**654**	**626**	**673**
	UNSAT	**625**	**553**	**562**
	BOTH	**1279**	**1179**	**1235**

Table 3.2: 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.

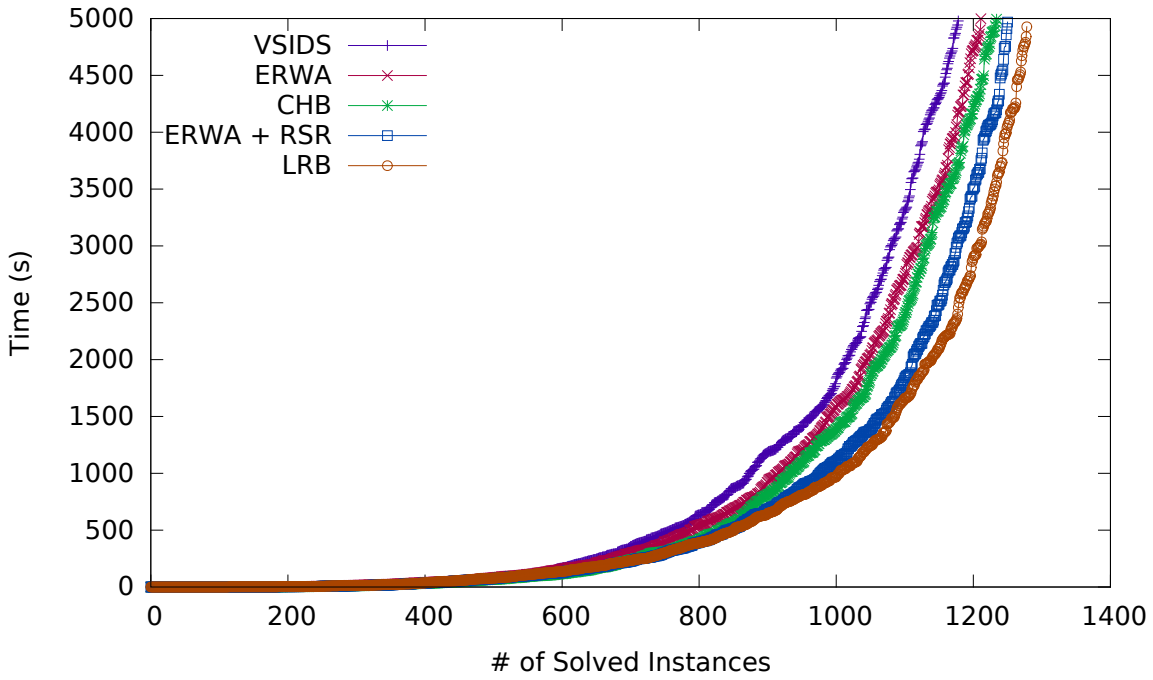

Figure 3.3: A cactus plot of the 5 branching heuristics in MiniSat 2.2 with aggressive LBD-based clause deletion. The benchmark consists of the 4 most recent SAT Competition benchmarks (2014, 2013, 2011, 2009) including both the application and hard combinatorial categories, excluding duplicate instances.

fact that VSIDS and CHB are designed without LR as an explicit objective. This perhaps partly explains the effectiveness of those two heuristics.

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 [80], Glucose [12], and Lingeling [14] which all implement VSIDS. We modified CryptoMiniSat 4.5.3 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 3.4. LRB improved CryptoMiniSat on 6 of the 8 benchmarks and solves 59 more instances overall.

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 (w/o 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**

Table 3.3: 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	CMS w/ LRB	CMS w/ 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 (w/o duplicates)	**SAT**	**619**	**598**	**575**	**589**
	UNSAT	**738**	**700**	**685**	**782**
	BOTH	**1357**	**1298**	**1260**	**1371**

Table 3.4: 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.

3.4 Stochastic Gradient Descent Branching Heuristic

GGB is too expensive in practice because of the computational cost of computing the c function. As mentioned earlier, branching heuristics need to balance the trade-off between selecting good decision variables and branching quickly. Unfortunately, GGB fails in the latter. In this section, we describe a new branching heuristic called the *stochastic gradient descent branching (SGDB)* heuristic that solves this issue by cheaply approximating $c : PA \to \{1, 0\}$ via machine learning.

The c function is a classifier that classifies partial assignments into either class 1 or 0. We approximate c by modelling it as a classification problem in machine learning. We opt for the popular 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. We use online stochastic gradient descent to train the function $\tilde{c}$ to approximate the function c during the search. 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 in logistic regression, $\tilde{c} := \sigma(w_0 + w_1x_1 + w_2x_2 + ... + w_nx_n)$ 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 by concatenating the literals on the conflict side of conflict analysis with the negation[1] 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 decision level. Hence PA_0 is given the label 0. In summary, SGDB creates two data points at every conflict, one for each class (the

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

conflict class and the non-conflict class) guaranteeing a balance between the two classes.

Two data points are created at every conflict. SGDB then applies one step of stochastic gradient descent on these two data points to update the weights after each conflict. 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. In other words, the weights of unassigned variables do not need to be updated immediately. 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 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 polarity 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:

$$
\begin{aligned}
&argmax_{v \in UnassignedVars} \tilde{c}(CPA \cup PhaseSaving(v)) \\
&= argmax_{v \in UnassignedVars} \sigma(w_0 + w_v + \sum_{l \in vars(CPA)} w_l) \\
&\text{Note that } \sigma \text{ is a monotonically increasing function.} \\
&= argmax_{v \in UnassignedVars} (w_0 + w_v + \sum_{l \in vars(CPA)} w_l) \\
&\text{Remove the terms common to all the iterations of argmax.} \\
&= argmax_{v \in UnassignedVars} w_v
\end{aligned}
$$

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 6.

Differences with VSIDS: The SGDB branching heuristic presented thus far has many similarities with VSIDS. During each conflict, VSIDS increments the activities of the

Algorithm 6 Pseudocode for the *SGDB* heuristic.

function PHASESAVING(Var) ▷ return the variable plus a polarity
  return $mkLit(Var, Var_{SavedPolarity})$
end function

procedure INITIALIZE
  for all $v \in Vars$ **do**
    $\alpha \leftarrow 0.8$, $\lambda \leftarrow 0.1 \times \alpha$, $w_v \leftarrow 0$
    $r_v \leftarrow 0$ ▷ Stores the last time v was lazily regularized.
  end for
  $conflicts \leftarrow 0$ ▷ The number of conflicts occurred so far.
end procedure

function GETPA1($learntClause$, $conflictSide$)
  return $\{\neg l \mid l \in learntClause\} \cup conflictSide$
end function

function GETPA0(PA_1)
  return $\{v \in AssignedVars \mid DecisionLevel(v) < currentDecisionLevel\} \setminus PA_1$
end function

procedure AFTERCONFLICTANALYSIS($learntClause$, $conflictSide$) ▷ Called after a learnt clause is generated from conflict analysis.
  if $\alpha > 0.12$ **then**
    $\alpha \leftarrow \alpha - 2 \times 10^{-6}$, $\lambda \leftarrow 0.1 \times \alpha$
  end if
  $conflicts \leftarrow conflicts + 1$
  $PA_1 \leftarrow GetPA1(learntClause, conflictSide)$
  $PA_0 \leftarrow GetPA0(PA_1)$
  for all $v \in vars(PA_1 \cup PA_0)$ **do** ▷ Lazy regularization.
    if $conflicts - r_v > 1$ **then**
      $w_v \leftarrow w_v \times (1 - \frac{\alpha\lambda}{2})^{conflicts - r_v - 1}$
    end if
    $r_v \leftarrow conflicts$
  end for
  $error_1 \leftarrow \sigma(w_0 + \sum_{i \in vars(PA_1)} w_i)$ ▷ Compute the gradients and descend.
  $error_0 \leftarrow \sigma(w_0 + \sum_{i \in vars(PA_0)} w_i)$
  $w_0 \leftarrow w_0 \times (1 - \frac{\alpha\lambda}{2}) - \frac{\alpha}{2}(error_1 + error_2)$
  for all $v \in vars(PA_1)$ **do**
    $w_v \leftarrow w_v \times (1 - \frac{\alpha\lambda}{2}) - \frac{\alpha}{2}(error_1)$
  end for
  for all $v \in vars(PA_0)$ **do**
    $w_v \leftarrow w_v \times (1 - \frac{\alpha\lambda}{2}) - \frac{\alpha}{2}(error_0)$
  end for
end procedure

function SGDB
  $d \leftarrow argmax_{v \in UnassignedVars} w_v$
  while $conflicts - r_d > 0$ **do** ▷ Lazy regularization.
    $w_d \leftarrow w_d \times (1 - \frac{\alpha\lambda}{2})^{conflicts - r_d}$
    $r_d \leftarrow conflicts$
  end while
end function

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 described in Algorithm 6 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 7 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 them to PA_1. This lets the learning algorithm associate these variables with the conflict class. See Algorithm 8 for the pseudocode.

Algorithm 7 Pseudocode for the sparse non-conflict extension. Only the `GetPA0` code is modified, the rest remains the same as SGDB.

function SAMPLE($level$)
    $C \leftarrow \{v \in Vars \mid DecisionLevel(v) = level\}$
    return a variable sampled uniformly at random from C
end function

function GETPA0(PA_1)
    return $(\bigcup_{i \in \{1,2,\ldots,currentDecisionLevel-1\}} Sample(i)) \setminus PA_1$
end function

Algorithm 8 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** $\{\neg l \mid l \in learntClause\} \cup conflictSide \cup adjacent$
4: **end function**

3.4.1 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 3.5 and Figure 3.4 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, and in total solve just 12 instances fewer than VSIDS. LRB solves 93 more instances than VSIDS. Table 3.6 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.

3.5 Related Work

The Chaff solver introduced the VSIDS branching heuristic in 2001 [68]. Although many branching heuristics have been proposed [40, 17, 78, 38, 65, 51], 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 [25]. 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. Loth et al. used multi-armed bandits for directing the growth of the search tree for Monte-Carlo

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

Table 3.5: # of solved instances by various configurations of SGD, VSIDS, and LRB.

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

Table 3.6: GLR and average LBD of various configurations of SGD, VSIDS, and LRB on the entire benchmark with duplicate instances removed. LRB solves the most instances and achieves the highest GLR and lowest average LBD in our experiments.

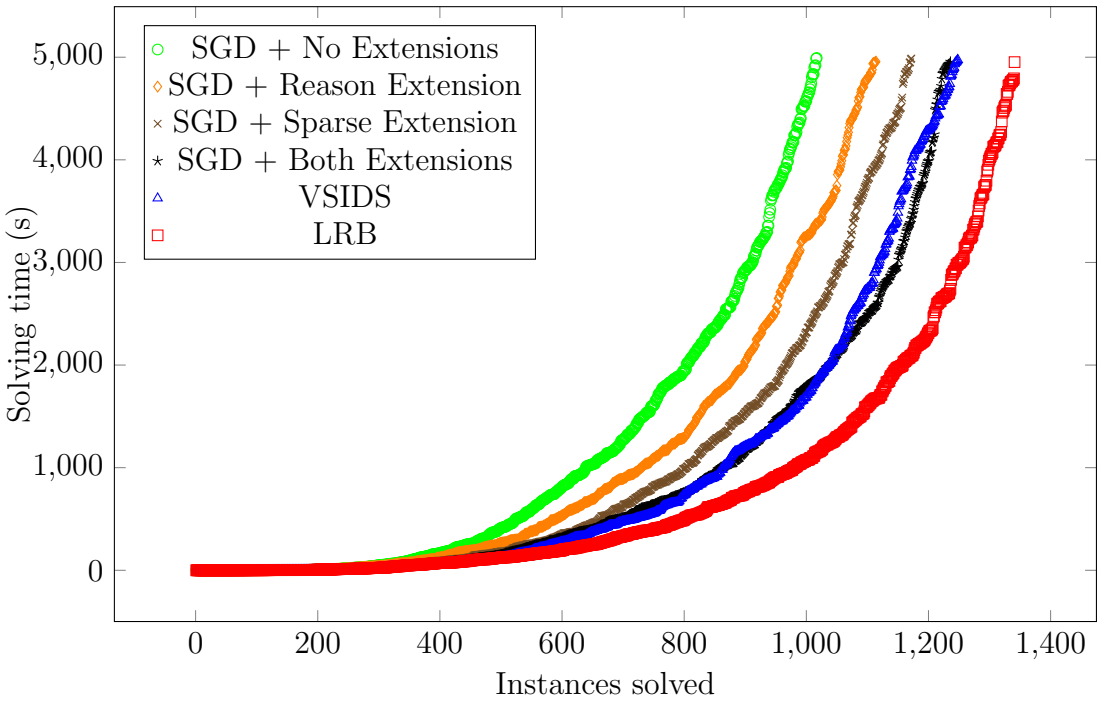

Figure 3.4: A cactus plot of various configurations of SGD, VSIDS, and LRB on the entire benchmark with duplicate instances removed.

Tree Search [62]. 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 [37]. The rewards they are optimizing is to minimize the number of unsatisfied clauses. Lagoudakis and Littman used reinforcement learning to dynamically switch between a fixed set of 7 well-known SAT branching heuristics [58]. Their technique requires offline training on a class of similar instances. Our technique differs in that machine learning selects the variables themselves, rather than selecting from a branching heuristic from a fixed set of predetermined heuristics.

Chapter 4

Understanding Branching Heuristics

Pipatsrisawat and Darwiche made a landmark discovery in 2009. They showed remarkably that CDCL solvers with nondeterministic branching and restarts p-simulate general resolution [75]. That is, problems with short general resolution proofs are tractable for CDCL solvers assuming the branching heuristic and the restart policy makes the correct choices. This goes to show the importance of the branching heuristic to a SAT solver in theory. On the empirical side, the story is the same, the branching heuristic is critically important for the awesome performance of CDCL SAT solvers we see today [52]. The shortcoming in practice is that the branching heuristic cannot simulate the nondeterministic branching order without knowing the proof of unsatisfiability a priori. Yet somehow the branching heuristic still needs to derive a good branching order somehow. This chapter is devoted to understanding how good branching heuristics arrive at a good order by studying their behavior.

The VSIDS branching heuristic is of primary interest due to its role in revolutionizing CDCL SAT solvers. Although VSIDS is just a composition of two simple procedures called bump and decay, its uncanny ability to conjure up good variable orderings for a wide range of instances is an enigma. In this chapter, we study the bump and decay components separately and understand them individually before treating VSIDS as a whole.

In this chapter, we first justify GLR as a good objective for branching by studying extant branching heuristics in SAT history. We see that the better heuristics tend to be those that achieve higher GLR. Second, we show that VSIDS bump is biased towards variables that are likely to increase GLR. However, this bump "signal" is noisy. Third, we show that VSIDS decay is an exponential moving average for smoothing out noise in a noisy signal. Hence the combination of bump and decay allows VSIDS to focus on the

variables most likely to increase GLR. Lastly, we show that the bump and decay can be viewed as finding constrained variables in a graphical CNF. More precisely, the variables ranked highly by VSIDS correlates with the variables ranked highly by graph centrality measures.

4.1 Understanding GLR

In Hypothesis 1, we conjectured that increasing GLR is good for branching. Our experimental results with GGB and LRB are consistent with that hypothesis. In this section, we conduct a more thorough experiment to support the hypothesis.

We test 7 well-known branching heuristics: LRB [60], CHB [59], VSIDS (MiniSat [32] variation of VSIDS), CVSIDS (Chaff [68] variation of VSIDS), Berkmin [40], DLIS [65], and Jeroslow-Wang [51]. We created 7 versions of MapleSAT [3], one for each branching heuristic, keeping the code unrelated to the branching heuristic untouched. We ran all 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. This experiment was conducted on StarExec. 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 [4].

The results are presented in Table 4.1. 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 almost 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 [81] (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 [35] 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,

Table 4.1: 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

Table 4.2: 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 [32] with phase saving [74], Luby restarts [63], and rapid clause deletion [10] based on LBD [10]. 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

a high (resp. low) GLR tends to have low (resp. high) solving time as we hypothesized. Table 4.2 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.

The hypothesis that GLR correlates with the solving time of various branching heuristic holds empirically for the set of well-known branching heuristics. Perhaps this should not come as a surprise given that the recent trend in branching heuristic design is to prioritize variables that caused conflicts, presumably assuming these variables continue to cause conflicts in the near future. We dive deeper in the next section on how the VSIDS branching heuristic accomplishes exactly that.

4.2 Understanding VSIDS

Various branching heuristics have come and gone. Over time, the *variable state independent decaying sum* (VSIDS) branching heuristic [68] has become the dominant branching heuristic for CDCL SAT solvers for over a decade. Understanding why VSIDS works well in practice is crucial in designing new heuristics. VSIDS is implemented as follows.

Activity: VSIDS associates a floating-point number for each variable called *activity*. Intuitively, the activity measures a variable's frequency of appearing in recent conflicts via the *bump* and *decay*. When it comes time to branch, VSIDS selects the unassigned variable with the highest activity. This is made easy by storing the activities in a priority queue. Typically activities are initialized to zero at the start of the search.

Bump: During conflict analysis, all the variables appearing in either the conflict side of the cut or the learnt clause have their activity additively increased by one, also called a *bump*. The bump (without the decay) is essentially counting the number of times each variable appears in either the conflict side of the cut or the learnt clause. We define a notable variation of VSIDS, which we refer to as cVSIDS, that only bumps variables in the learnt clause. cVSIDS bumps a subset of variables that VSIDS bumps, and they perform decay in the same manner.

Decay: After each conflict, the activities of all variables is multiplicatively decreased by a decay factor α where $0 < \alpha < 1$. A naive implementation of decay takes linear time to perform the multiplication for each variable activity. MiniSat introduced a clever implementation of decay [32] that reduces the cost to amortized constant time.

It is beneficial to view the bumps of variable v as a time series b_v defined as follows.

$$b_v := \langle b_{v,1}, b_{v,2}, ..., b_{v,t} \rangle \tag{4.1}$$

$$b_{v,i} := \begin{cases} 1 & \text{if variable } v \text{ is bumped on the } i^{th} \text{ conflict} \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

Then the activity of variable v at time the t^{th} conflict is defined recursively as follows.

$$activity_{v,t} = \begin{cases} 0 & \text{if } t = 0 \\ \alpha(b_{v,t} + activity_{v,t-1}) & \text{otherwise} \end{cases} \tag{4.3}$$

Adding $b_{v,t}$ to the previous activity is the bump and multiplying by α is the decay. In this definition, we define the initial activity as zero which is common. Setting the initial activity is called *initialization*, and the activity values chosen for initialization can drastically alter the running time. For now, we ignore initialization and focus on the fundamentals, the bump and decay. The key to understanding VSIDS is to address the following two questions.

1. What is special about the class of variables that VSIDS chooses to additively bump?
2. What role does multiplicative decay play in making VSIDS so effective?

We will show that the answer to the first question is that VSIDS bumps variables that are likely to cause conflicts and hence increase GLR if branched on. The answer to the second question is that the multiplicative decay behaves like an exponential moving average (EMA) that favors variables that persistently occur in conflicts (the signal) over variables that occur intermittently (the noise). The combination of these two answers gives VSIDS the power to realize variable orderings that achieve high GLR.

4.2.1 Understanding VSIDS Bump

Without the VSIDS bump, the variable activities will remain zero. But what is special about the set of variables VSIDS chooses to additively bump? The GLR experiment in Table 4.1 provides a good hint. VSIDS achieves high GLR by branching on variables that are more likely to cause conflicts in the future. Paired with the fact that it bumps variables that caused conflicts in the past, it seems natural to speculate that VSIDS bumps variables that cause GLR to go up if branched on.

Recall the definition of function $c : PA \to \{1, 0\}$ from equation 3.1. 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".

The process of branching appends a variable assignment to the current partial assignment to create a new partial assignment PA'. If $c(PA') = 1$, then the GLR increases since

a conflict is imminent once BCP is invoked. The most recent decision caused the GLR to increase. Otherwise $c(PA') = 0$ means the GLR decreases since a decision was made but no conflict occurs. The idea is to use c to classify all the potential decision variables to either class 1 (i.e., those that increase GLR if branched on) and class 0 (i.e., those that decrease GLR if branched on). See Algorithm 9 for the pseudocode. If increasing GLR is the goal, then VSIDS should bump variables in class 1 to increase its likelihood of branching on these variables to obtain higher GLR.

Algorithm 9 Pseudocode for classifying a potential decision variable x for CNF ϕ.

```
1: function PhaseSaving(Var)                    ▷ Return the variable plus a sign.
2:     return mkLit(Var, Var_savedPolarity)
3: end function
4:
5: function Classify(x)
6:     CPA ← CurrentPartialAssignment
7:     PA' ← CPA ∪ PhaseSaving(x)
8:     return c(PA')
9: end function
```

Hypothesis 2 *The set of variables VSIDS chooses to bump has an overrepresentation of variables in class 1 (the conflict class) relative to the general population of variables.*

In this experiment, we ran Glucose 4.1 [1] over all 350 instances in the main track from the SAT Competition 2017. Each instance is given a timeout of 6 hours on StarExec [82]. Whenever a conflict occurs, we perform the following steps.

1. Record the set of variables bumped by VSIDS.
2. Allow the solver to resolve the conflict by learning a new clause, backjumping, and propagating the new learnt clause.
3. For all potential decision variables (i.e., all the unassigned variables), we classify them as either class 1 or class 0.
4. If none of the variables have class 1, then we ignore this conflict. Otherwise proceed to the next step.

[1] Glucose is a popular and competitive CDCL SAT solver often used in experiments because of its efficacy and simplicity (http://www.labri.fr/perso/lsimon/glucose/)

5. We record the following information:

   - BumpedClassOne: the number of potential decision variables that were bumped by the conflict and are in class 1.
   - Bumped: the number of potential decision variables that were bumped by the conflict.
   - ClassOne: the number of potential decision variables that are in class 1.
   - All: the number of potential decision variables.

At the end of the run, we compute the following two numbers for each instance.

$$B = \frac{\sum BumpedClassOne}{\sum Bumped} \tag{4.4}$$

$$C = \frac{\sum ClassOne}{\sum All} \tag{4.5}$$

B represents the proportion of bumped potential decision variables that were in class 1. If B is relatively high, then the VSIDS bump is targeting class 1 variables. This biases VSIDS to branch on these variables thus increasing GLR. C represents the percentage of potential decision variables that are in class 1. C describes the proportion class 1 variables occurring in the general population of variables. We then compute the *bump-conflict ratio* defined as $\frac{B}{C}$. A high ratio means that VSIDS bump targets class 1 variables disproportionately. We record the bump-conflict ratio for every instance and report the ratios in Figure 4.1 in the form of a histogram. The median bump-conflict ratio is 11.75 and the average bump-conflict ratio is 422.40. So for most instances, the bumped variable is over 11 times more likely to be in class 1 than the average variable. Only 5 of the 350 instances recorded a ratio less than 1, with the minimum ratio being 0.953.

Understanding VSIDS bump helped with our branching heuristic work. Since VSIDS uses conflict analysis to identify variables likely to cause conflicts, we used the same clause analysis mechanism as a reward signal in the reinforcement learning context.

We provide an complementary explanation for the set of variables VSIDS chooses to bump. Recall that a 1-UIP learnt clause is derived by a sequence of resolution rule applications starting from the conflicting clause. In this sequence of resolutions, only the final resolvent (i.e., the 1-UIP learnt clause itself) is stored in the learnt clause database. The intermediate clauses in this sequence of resolutions are discarded. The 1-UIP learnt

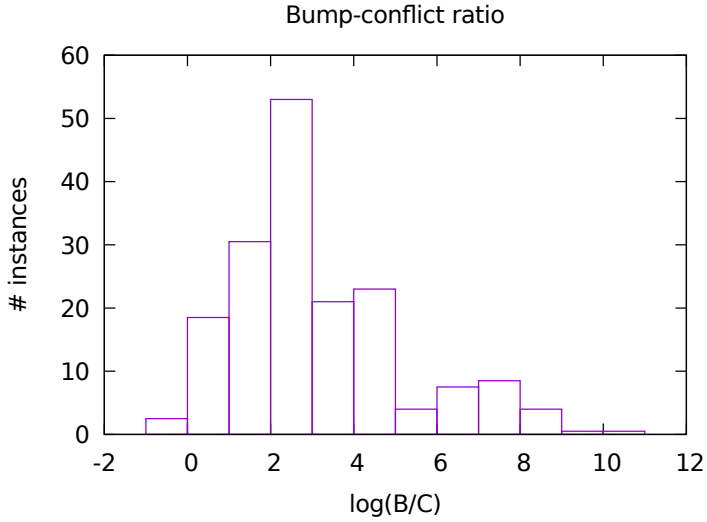

Figure 4.1: Histogram of the log bump-conflict ratio.

clause is beneficial because it is 1-empowering, but many of these intermediate clauses are also 1-empowering [76]. We conjecture that many of these intermediate clauses remain 1-empowering even after learning the 1-UIP clause. So we want to branch on the variables corresponding to these 1-empowering intermediate clauses to hopefully absorb them through conflicts. That is, to learn additional clauses so that they no longer remain 1-empowering. The variables corresponding to these 1-empowering intermediate clauses are exactly the variables on the conflict side, which VSIDS bumps.

Hypothesis 3 *Intermediate clauses of the 1-UIP learning scheme are mostly 1-empowering even after adding the 1-UIP learnt clause to the learnt clause database.*

In this experiment, we ran Glucose 4.1 over all 350 instances in the main track from the SAT Competition 2017. Each instance is given a timeout of 6 hours on StarExec [82]. Whenever a conflict occurs, we record all the intermediate clauses. We then allow the solver to learn the 1-UIP clause. After, we scan the intermediate clauses from the most recent conflict and mark whether they are 1-empowering or not. At the end of the run, we report the fraction between the total number of marked intermediate clauses over the total number of intermediate clauses for every instance. We report all the fractions in Figure 4.2. A fraction of 1 (resp. 0) means all (resp. none) of the intermediate clauses remain 1-empowering after the 1-UIP clause is learnt. The median fraction is 0.685 and

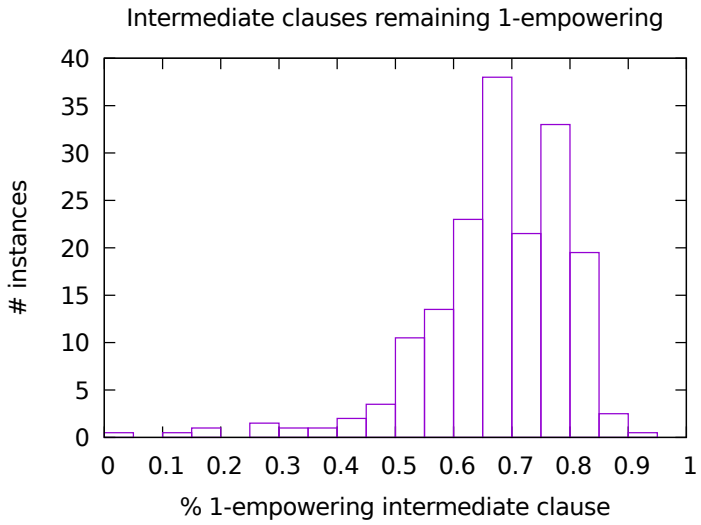

Figure 4.2: Histogram of the percentage of intermediate clauses that remain 1-empowering after the 1-UIP clause is learnt.

the average fraction is 0.676. So approximately 67% of the intermediate clauses are not absorbed and benefit from having their variables bumped.

4.2.2 Understanding VSIDS Decay

The inclusion of multiplicative decay in VSIDS gives it the distinctive feature of focusing its search based on recent conflicts. Figure 4.3 shows the importance of decay in VSIDS from a performance perspective. Without decay, Glucose solved 57 fewer instances out of the 350 in the benchmark.

The original Chaff paper [68] and patent [67] motivates the decay by rather cryptically mentioned that VSIDS acts like a "low-pass filter" without specifying what signals are being fed to this filter, and why the high-frequency components are being filtered out and discarded. While Huang et al. [48] referred to VSIDS as an EMA, we will show this explicitly. On top of explicitly characterizing VSIDS as an EMA, we also describe how this plays a crucial role in the effectiveness of VSIDS as a branching heuristic. We first reformulate how activities are calculated to better highlight the relationship between VSIDS and EMA.

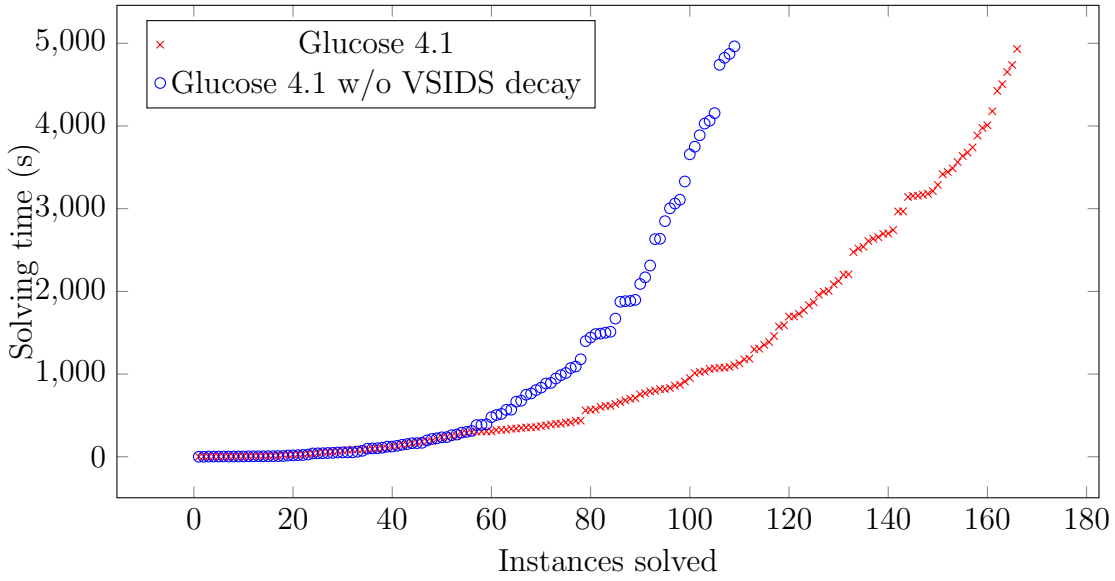

Figure 4.3: Cactus plot of Glucose 4.1 with and without VSIDS decay on 350 instances from the main track of SAT Competition 2017.

$$activity_{v,t} = \begin{cases} 0 & \text{if } t = 0 \\ \alpha(b_{v,t} + activity_{v,t-1}) & \text{otherwise} \end{cases} \tag{4.6}$$

$$= \begin{cases} 0 & \text{if } t = 0 \\ \alpha b_{v,t} + \alpha activity_{v,t-1} & \text{otherwise} \end{cases} \tag{4.7}$$

$$= \begin{cases} 0 & \text{if } t = 0 \\ (1-\alpha)(\frac{\alpha}{1-\alpha} b_{v,t}) + \alpha activity_{v,t-1} & \text{otherwise} \end{cases} \tag{4.8}$$

In the last equation, the recursive definition of activity is remarkably similar to the recursive definition of EMA. We define a new time series b' for bumps that scales all the elements of b by $\frac{\alpha}{1-\alpha}$.

$$b'_v := \langle \frac{\alpha}{1-\alpha} b_{v,1}, \frac{\alpha}{1-\alpha} b_{v,2}, ..., \frac{\alpha}{1-\alpha} b_{v,t} \rangle \tag{4.9}$$

Then the equation of activity can be further simplified.

$$activity_{v,t} = \begin{cases} 0 & \text{if } t = 0 \\ (1-\alpha)(\frac{\alpha}{1-\alpha} b_{v,t}) + \alpha activity_{v,t-1} & \text{otherwise} \end{cases} \tag{4.10}$$

$$= \begin{cases} 0 & \text{if } t = 0 \\ (1-\alpha) b'_{v,t} + \alpha activity_{v,t-1} & \text{otherwise} \end{cases} \tag{4.11}$$

$$= EMA_{1-\alpha}(b'_v) \tag{4.12}$$

In this final equation, it is clear that VSIDS activity is exactly an EMA over the scaled bump time series b'. The EMA causes VSIDS to favor variables that "persistently" occur in "recent" conflicts. Recall that earlier we showed that VSIDS bump targets class 1 variables. However this data is extremely noisy, and not always accurate. Additionally, the set of class 1 variables changes over time as the current partial assignment morphs and the learnt clause database expands. Hence there is a temporal aspect to understanding the bump signals. For these reasons, an EMA helps reduce the noise and handle the temporality of class 1 variables. As a result, adding decay to VSIDS greatly increases the GLR since it is better able to single out class 1 variables than without decay.

Understanding VSIDS decay was greatly beneficial to our branching heuristic work. The VSIDS connection with EMA led us directly to ERWA in the reinforcement learning branching context since it also employs EMA to smooth out noise and handle temporality.

4.2.3 Correlation Between VSIDS and Graph Structure

Variables form logical relationships and we hypothesize that VSIDS exploit these relationships to find the variables that are most "constrained" in the formula. The logical relationship between variables are concretized as some variation of the variable incidence graph (VIG). We then use various metrics from graph theory to find the "constrained" variables and show that VSIDS concentrates on these variables.

VSIDS Favors Bridge Variables

Researchers have found that SAT formulas encountered in the real world tend to exhibit community structure [7]. Shortly after, a different group of researchers show that formulas with good community structure tend to be easier to solve [72]. It was natural for us ask

the question whether VSIDS somehow exploits the community structure of SAT instances thus formulas with good community structure are more suitable for modern solvers.

What we find in practice is that if we singled out a single community and erased the other communities from the formula, the SAT solver solves the single community very quickly. Bridge variables "constrain" the formula in the sense that each community is easy to solve individually but the bridge variables add dependencies between the communities which makes the entire problem difficult to solve. The connection between community structure and solving time is perhaps that good community structure lends itself to divide-and-conquer because the bridges are easier to satisfy. More precisely, the solver can focus its attention on the bridges by picking the bridge variables and assigning them appropriate values. When it eventually assigns the correct values to enough bridges, the VIG is separated among the community boundaries, and each partition can be solved with no interference from each other. Even if the VIG cannot be completely separated, it may still be beneficial to the cut bridges between communities so that these communities can be solved relatively independently. This analysis is consistent with a technique proposed in 2015 [8] to take advantage of community structure. In that paper [8], the authors propose a preprocessing technique of solving pairs of communities by severing the bridges with other communities to produce learnt clauses. Their work assumes that after severing the bridges, the communities are easy to solve hence their preprocessing technique is cheap. Additionally, they assume that the learnt clause quality improves after severing the bridges, hence their technique produces clauses that are better than just running the CDCL SAT solver on the original formula with all the communities intact. CNF inputs with good community structure by definition have fewer bridges, hence fewer bridges to cut before separation in the VIG occurs. Based on this intuition, we propose the following hypothesis.

Hypothesis 4 *VSIDS disproportionately picks, bumps, and learns clauses over the bridge variables in the community structure of SAT instances.*

VSIDS bumping and picking bridge variables lends itself to finding the correct value of bridge variables to separate the VIG. Learning clauses over bridge variables help correct wrong assignments to bridge variables.

To support the hypothesis, we performed the following experiments over the 1030 instances from SAT Competition 2013, after simplification using the MiniSat [32] simplifying-solver. We used the Louvain method [18] to compute the communities of the VIG of the input SAT formulas. We picked Louvain because it scales well with the size of input graphs. For each instance, the Louvain method is given an hour to compute and save the best community partition it finds. The community information is then given to a modified

MiniSat 2.2.0 so it can track the bridge variables. Due to the high cost, we only compute the communities once at the start. The modified MiniSat is given a timeout of 5000 seconds per instance, as per the SAT Competition 2013 rules. Before MiniSat begins its CDCL loop, it reads in the community information stored by the Louvain method. The solver then scans through its the initial input clauses and checks which variables share at least one clause with another variable residing in a different community and marks them as bridge variables. Our modified version of MiniSat checks if variable x is a bridge variable whenever one of 3 conditions occur.

1. Picks a decision variable x.
2. Bumps a variable x.
3. Learns a clause over variable x during the search.

If the variable is a bridge, the solver updates its internal counters to keep track of the number of bridge variables in the each of the 3 scenarios. At the end of the run, the solver outputs the percentage of variables that are bridge in each of these scenarios. This additional code adds little overhead and does not change the behavior of MiniSat. We are simply instrumenting the solver to collect statistics of interest with the community computation performed beforehand.

For each instance, we compute the percentage of decision variables, bumped variables, learnt clause variables, and number of variables in the CNF that are also bridges. Then we averaged these percentages over the three SAT 2013 Competition benchmark categories (application, combinatorial, and random) and reported these numbers.

Table 4.3 shows that bridge variables are highly favored in MiniSat by its branching heuristic, conflict analysis, and clause-learning. It is a surprising result that bridge variables are favored even though the heuristics and techniques in MiniSat have no notion of communities. While bridge variables certainly make up a large percent of variables, the percent of picked bridge variables is even higher. Table 4.3 includes only the instances where the Louvain implementation completed before timing out. In total, 229/300 instances in the application category and 238/300 instances in the hard combinatorial category are included. In the random category, every variable is a bridge, hence the results are omitted. This is expected because it is highly improbable to generate random instances where a variable is not neighboring another variable outside its community.

Category	% of variables that are bridge	% of picked variables that are bridge	% of bumped variables that are bridge	% of learnt clause variables that are bridge
Application	61.0	79.9	71.6	78.4
Combinatorial	78.2	87.6	84.3	88.2

Table 4.3: MiniSat's VSIDS and clause learning prefers to pick, bump, and learn over bridge variables.

VSIDS Favors High Temporal Centrality Variables

Centrality measures the important vertices in a graph and is a contender for the definition of "constrainedness" of variables. A variable with high degree centrality appears in many clauses, hence it is highly "constrained" in the formula since every clause constrains the valid assignments to that variable. Eigenvector centrality defines constrainedness recursively. A variable with high eigenvector centrality appears in many clauses containing variables with high eigenvector centrality. That is, it is not enough for a variable to be in many clauses for it to be "constrained" according to eigenvector centrality, those clauses must also contain highly constrained variables. In this section, we describe the experiments to support the following hypothesis.

Hypothesis 5 *VSIDS and cVSIDS, viewed as ranking functions, correlate strongly with both temporal degree centrality and temporal eigenvector centrality.*

Recall that VSIDS decay implements an EMA over the bump time series which gives us a clue that the temporal aspect of clause learning needs to be taken into account. To incorporate the temporal aspect of learnt clauses we introduce *temporal variable incidence graph* (TVIG) here, that extends the VIG by encoding temporal information into its structure. During the CDCL search, let t_{now} be the number of conflicts that has occurred thus far. Let $t(C)$ be the number of conflicts that has occurred up to learning clause C. In the TVIG, every clause is labeled with a timestamp denoted $timestamp(C)$.

$$timestamp(C) := \begin{cases} 0 & \text{if clause } C \text{ is from the original input formula} \\ t(C) & \text{if clause } C \text{ is a learnt clause} \end{cases} \tag{4.13}$$

Fix an *exponential smoothing factor* $0 < \alpha < 1$. The TVIG is a weighted graph constructed in the same manner as the VIG except the weight function W is defined differently.

$$W(xy) = \sum_{C \in \phi \wedge x \in C \wedge y \in C \wedge x \neq y} \frac{\alpha^{t_{now} - timestamp(C)}}{|C| - 1} \tag{4.14}$$

The only difference in the weight function between the TVIG and VIG is that the weights are scaled by the factor $\alpha^{t_{now} - timestamp(C)}$. The exponent $t_{now} - timestamp(C)$ is interpreted as the *age* of the clause. Since $timestamp(C)$ is constant and t_{now} increments after ever conflict, the age of a clause keeps increasing. Observe that the TVIG evolves throughout the solving process: as new learnt clauses are added, new edges are added to the graph, and the ages of extant edges increase. As an edge's age increases, its weight decreases exponentially with time assuming no new learnt clause contains its pair of variables. The exponential decrease in weights is inspired by VSIDS decay which is an EMA as shown earlier in Section 4.2.2. In many domains, it is often the case that more recent data points are more useful than older data points. We then define two *temporal graph centrality* (TGC) measures over the TVIG called the *temporal degree centrality* (TDC) and *temporal eigenvector centrality* (TEC).

Definition 6 (Temporal Degree Centrality (TDC)) *The temporal degree centrality of a variable is the degree centrality of the variable's corresponding vertex in the TVIG.*

Definition 7 (Temporal Eigenvector Centrality (TEC)) *The temporal eigenvector centrality of a variable is the eigenvector centrality of the variable's corresponding vertex in the TVIG.*

We use two measurements to show that VSIDS correlates with TDC and TEC, the Spearman's rank correlation coefficient and top-k.

Spearman's Rank Correlation Coefficient: At any time during the search, we can rank the variables in descending order based on variable activities. We can construct a second ranking of variables in descending order based on their TGC. We then compute the Spearman's rank correlation coefficient [81] between the VSIDS and TGC rankings. Spearman's rank correlation coefficient is a widely-used correlation coefficient in statistics for measuring the degree of relationship between a pair of rankings. The strength of Spearman's correlation is conventionally interpreted as follows: 0.00–0.19 is very weak, 0.20–0.39 is weak, 0.40–0.59 is moderate, 0.60–0.79 is strong, 0.80–1.00 is very strong. We follow the standard practice of applying the Fisher transformation [35] when computing the average of correlations.

Top-k: Let v be the unassigned variable with the highest ranked according to some VSIDS variant. Let i be the position of variable v according to a specific TGC ranking, excluding assigned variables. Then the *top-k measure* is 1 if $i \leq k$, otherwise 0. The rationale for this metric is that SAT solvers typically only choose the top-ranked unassigned variable, according to the VSIDS ranking, to branch on. The ordering of the lower ranked variables is mostly inconsequential in practice, yet the Spearman correlation considers the proper ordering of the lower ranked variables just as important as the higher ranked variables. If the VSIDS top-ranked unassigned variable occurs very often among the top-k ranked variables according to TGC, then we infer that VSIDS picks variables that are highly ranked according to TGC. In our experiments, we used various values for k.

We perform the following experiment to support that VSIDS correlates with TDC and TEC. We implemented the VSIDS variants and TGC measures in MiniSat 2.2.0 [32]. All the experiments were performed using MiniSat on all 1030 Boolean formulas obtained from all three categories (application, combinatorial, and random) of the SAT Competition 2013. Before beginning any experimentation, the instances are first simplified using MiniSat's inbuilt preprocessor with the default settings. All experiments were performed on the SHARCNET cloud [2], where cores range in specs between 2.2 to 2.7 GHz with 4 GB of memory, and 4 hour timeout. We use 100 iterations of the power iteration algorithm [41] to compute TEC. We use MiniSat's default decay factor of 0.95 for VSIDS. We also use 0.95 as the exponential smoothing factor for the TVIG. We take Spearman correlation and top-k measurements on the current state of the solver after every 5000 iterations, where an iteration is defined as a decision or a conflict. Observe that we take measurements dynamically as the solver solves an instance, and not just once at the beginning. Such a dynamic comparison gives us a much better picture of the correlation between two different ranking functions or measures than a single point of comparison. After the solver terminates on an instance, we compute the average Spearman correlation and top-k measured during the run of the solver.

We measured the average Spearman correlation and top-k for all combinations of VSIDS/cVSIDS and TDC/TEC on every instance. We take all the instance averages and average them again, and report the average of the averages. The final numbers are labeled as "mean top-k" or "mean Spearman". For example, a mean top-10 of 0.912 is interpreted as "for the average instance in the experiment, 91.2% of the measured top-ranked variables according to VSIDS are among the 10 unassigned variables with the highest centrality". Likewise, a high mean Spearman implies the average instance has a strong positive correlation between VSIDS and TGC rankings.

	cVSIDS vs TDC			VSIDS vs TDC		
	Application	Combinatorial	Random	Application	Combinatorial	Random
Mean Spearman	0.818	0.946	0.988	0.629	0.791	0.864
Mean Top-1	0.884	0.865	0.949	0.427	0.391	0.469
Mean Top-10	0.912	0.898	0.981	0.705	0.735	0.867

Table 4.4: Results of comparing cVSIDS and VSIDS with TDC.

	cVSIDS vs TEC			VSIDS vs TEC		
	Application	Combinatorial	Random	Application	Combinatorial	Random
Mean Spearman	0.790	0.926	0.987	0.675	0.764	0.863
Mean Top-1	0.470	0.526	0.794	0.293	0.304	0.418
Mean Top-10	0.693	0.746	0.957	0.610	0.670	0.856

Table 4.5: Results of comparing cVSIDS and VSIDS with TEC.

The results are presented in Table 4.4 (resp. Table 4.5) comparing VSIDS and TDC (resp. TEC) rankings. The data shows a strong correlation between VSIDS variants and TDC, in particular, the 0.818 mean Spearman between cVSIDS and TDC is high. The metrics are lower with TEC, but the correlation remains strong. VSIDS has a better mean Spearman with TEC than TDC in the application category. We have also conducted this experiment with non-temporal degree/eigenvector centrality and the resulting mean Spearman and mean top-k are significantly lower than their temporal counterparts.

It is commonly believed that VSIDS focuses on the "most constrained part of the formula" [45], and that this behavior is responsible for its surprising effectiveness in practice. However, the term "most constrained part of the formula" has previously not been well-defined in a mathematically precise manner. One intuitive way to define the constrainedness of a variable is to analyze the Boolean formula, and count how many clauses a variable occurs in. The variables can then be ranked based on this measure. In fact, this measure is the basis of the branching heuristic called DLIS [65], and was once the dominant branching heuristic in SAT solvers. We show that graph centrality measures are a good way of mathematically defining this intuitive notion of syntactic "constrainedness of variables" that has been used by the designers of branching heuristics. Degree centrality of a vertex in the VIG is indeed equal to the number of clauses it belongs to, hence it is a good basis for guessing the constrained variables for the same reason. Eigenvector centrality extends this intuition by further increasing the ranks of variables close in proximity to other constrained variables in the VIG. Additionally, as the dynamic structure of the VIG evolves due to the addition of learnt clauses by the solver, the most highly constrained variables in a given instance also change over time. Hence we incorporated learnt clauses and temporal information into the TVIG to account for changes in variables'

constrainedness over time.

Besides the success of branching heuristics like VSIDS and DLIS, there is additional evidence that the syntactic structure is important for making good branching decisions. For example, Iser et al. discovered that initializing the VSIDS activity based on information computed on the abstract syntax tree of their translator has a positive impact on solving time [50]. In a different paper [72], the authors have shown that the graph-theoretic community structure strongly influences the running time of CDCL SAT solvers. This is more evidence of how CDCL SAT solver performance is influenced by syntactic graph properties of input formulas.

4.3 Related Work

Armin Biere [13] described the low-pass filter behavior of VSIDS, and Huang et al. [48] stated that VSIDS is essentially an EMA. Katsirelos and Simon [53] were the first to publish a connection between eigenvector centrality and branching heuristics. In their paper [53], the authors computed eigenvector centrality (via Google PageRank) only once on the original input clauses and showed that most of the decision variables have higher than average centrality. Also, it bears stressing that their definition of centrality is not temporal. By contrast, our results correlate VSIDS ranking with temporal degree and eigenvector centrality, and show the correlation holds dynamically throughout the run of the solver. Also, we noticed that the correlation is also significantly stronger after extending centrality with temporality. Simon and Katsirelos do hypothesize that VSIDS may be picking bridge variables (they call them fringe variables). However, they do not provide experimental evidence for this. To the best of our knowledge, we are the first to establish the following results regarding VSIDS.

1. VSIDS disproportionately bumps variables in class 1, the conflict class.
2. Explain the importance of EMA (multiplicative decay) to the effectiveness of VSIDS.
3. VSIDS picks, bumps, and learns high-centrality bridge variables.

Chapter 5

Machine Learning-Based Restart Policy

A CDCL SAT solver can be thought of as constructing a search tree on-the-fly exploring possible solutions. The solver frequently restarts, that is, it discards the current search tree and begins anew (but does not throw away the learnt clauses and the variable activities). Although this may seem counterproductive, SAT solvers that restart frequently are faster empirically than solvers that opt not to restart. Figure 5.1 shows the importance of restarts in CDCL SAT solvers. Without restarts, Glucose solved 15 fewer instances out of the 350 in the benchmark.

Researchers have proposed a variety of hypotheses to explain the connection between restarts and performance such as exploiting variance in the runtime distribution [63, 42] (similar to certain kinds of randomized algorithms). For various reasons however, we find these explanations for the power of restarts do not apply to the CDCL SAT solver setting. Instead, we take inspiration from Hamadi et al. who claim that the purpose of restarts is to compact the assignment stack [45], that is, the assignment stack is smaller with restarts enabled. We then further show that a compact stack tends to improve the quality of clauses learnt where we define quality in terms of the well-known metric *literal block distance* (LBD). Despite the search tree being discarded by a restart, learnt clauses are preserved so learning higher quality clauses continues to reap benefits across restarts. Learning higher quality clauses allows the solver to find a solution quicker. However, restarting too often incurs a high overhead of constantly rebuilding the search tree. So it is imperative to balance the restart frequency to improve the LBD but avoid excessive overhead incurred by restarts.

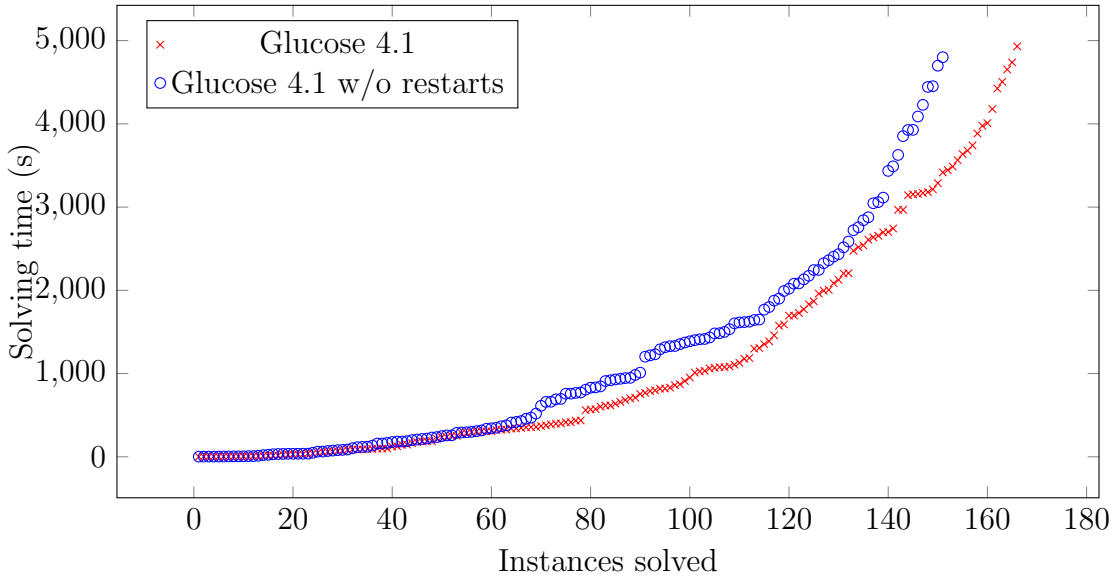

Figure 5.1: Cactus plot of Glucose 4.1 with and without restarts on 350 instances from the main track of SAT Competition 2017.

Based on these observations, we present a new machine learning-based restart policy in this chapter that is competitive with the state-of-the-art policies. Since restarts empirically reduce LBD, we designed a restart policy that tries to avoid high LBDs by restarting at the right times. Intuitively, our restart policy does the following: restart if the next learnt clause has an LBD in the 99.9^{th} percentile. Implementing this policy requires new techniques to answer the two following questions: is an LBD in the 99.9^{th} percentile and what is the LBD of the next learnt clause. We designed techniques to estimate answers to these two questions. Regarding the first question, the normal distribution is a good approximation for the right tail of the LBD distribution. The answer for the second question is to use machine learning to predict the LBD of the next learnt clause.

5.1 Prior Hypotheses on "The Power of Restarts"

In this section, we discuss prior hypotheses on the power of restarts in the DPLL and local search setting and their connection to restarts in the CDCL setting.

5.1.1 Heavy-tailed Distribution and Las Vegas Algorithm Hypotheses

From the perspective of Las Vegas algorithms, some researchers have proposed that restarts in CDCL SAT solvers take advantage of the variance in solving time [63, 42]. For a given input, the running time of a Las Vegas algorithm is characterized by a probability distribution, that is, depending on random chance the algorithm can terminate quickly or slowly relatively speaking. A solver can get unlucky and have an uncharacteristically long running time, in which case, a restart gives the solver a second chance of getting a short runtime [63]. It can continue to restart until it gets lucky.

Moreover, a heavy-tailed distribution was observed for various satisfiable instances on randomized DPLL SAT solvers [42]. Intuitively, this means that the probability of a long runtime is non-negligible in practice. Hence DPLL SAT solvers are likely to benefit from restarts assuming some randomization in the solver.

However, this explanation does not lift to restarts in modern CDCL solvers. First, most modern CDCL SAT solver implementations are not Las Vegas algorithms, that is, they are deterministic algorithms. Hence restarts cannot take advantage of variance in the solving time like in Las Vegas algorithms. Second, the optimal restart policy for Las Vegas algorithms has a restart interval greater than the expected solving time of the input [63]. So difficult instances should restart very infrequently. However in practice, even difficult instances with high solving time benefit from very frequent restarts in CDCL SAT solvers. Third, the definition of restarts in the context of Las Vegas algorithms differs significantly from the restarts implemented in CDCL SAT solvers. In Las Vegas algorithms, the restarts are equivalent to starting a new process, where the algorithm starts an new independent run from scratch preserving nothing from the previous run. Restarts in CDCL are only partial, the assignment stack is erased but everything else preserved (i.e., learnt clauses, saved phases, activity, etc.). Since the phases are saved, the CDCL SAT solver reassigns variables to the same value across restart boundaries [77]. As the authors of ManySAT [45] note: "Contrary to the common belief, restarts are not used to eliminate the heavy tailed phenomena since after restarting SAT solvers dive in the part of the search space that they just left." Fourth, the heavy-tailed phenomena was found to be true only for satisfiable instances, and yet empirically restarts are known to be even more relevant for unsatisfiable instances.

5.1.2 Escaping Local Minima Hypothesis

Another explanation for restarts comes from the context of optimization. Many optimization algorithms (in particular local search algorithms), get stuck in the local minima. Since local search only makes small moves at a time, it is unlikely to move out of a deep local minimum. The explanation is that restarts allow the optimization algorithm to escape the local minimum by randomly moving to another spot in the solution space. Certain local-search based SAT solvers with the objective to minimize the number of unsatisfied clauses perform restarts for this very purpose [47, 79]. However, restarts in CDCL do not behave in the same manner. Instead of setting the assignment of variables to random values like in local search, CDCL solvers revisit the same (or nearby) search space of assignments even after restarts since the variable activities and phases are preserved across restart boundaries [77].

As we show in Section 5.2, our hypothesis for the power of restarts is indeed consistent with the "escaping local minima" hypothesis. However, restarts enable CDCL solvers to escape local minima in a way that works differently from local search algorithms. Specifically, CDCL solvers with restarts enabled escape local minima by jumping to a nearby space to learn "better clauses", while local search algorithms escape local minima by randomly jumping to a different part of the search space.

5.2 "Restarts Enable Learning Better Clauses" Hypothesis

In this section, we propose that restarts enable a CDCL solver to learn better clauses. To justify our hypothesis, we start by examining the claim by Hamadi et al. [45] stating that "In SAT, restarts policies are used to compact the assignment stack and improve the order of assumptions." Recall that the only thing that changes during a restart in a CDCL solver is the assignment stack, and hence the benefits of restarts should be observable on the assignment stack. We show that this claim is empirically true by demonstrating that restarting frequently correlates with a compact assignment stack. We then go one step further to show that a compact assignment stack leads to better clause learning. More precisely, the solver ends up learning clauses with lower LBD, thereby supporting our hypothesis, and this in turn improves the solver performance.

Hypothesis 6 *Frequent restarts compact the assignment stack.*

Hypothesis 7 *Compacting the assignment stack lowers the LBD of learnt clauses.*

Hypothesis 8 *Lower LBD of learnt clauses correlates with better performance.*

Restarts do incur a cost though [77], for otherwise restart after every conflict would be the optimal policy for all inputs. After a solver restarts, it needs to make many decisions and propagations to rebuild the assignment stack from scratch. We call this the *rebuild time*. More precisely, whenever a solver performs a restart, we note the current time and the assignment stack size x right before the restart. Then the rebuild time is the time taken until either the solver encounters a new conflict or the new assignment stack size exceeds x through a series of decisions and propagations. Since we want to isolate the benefit of restart, we need to discount the cost of rebuilding. We define *effective time* to be the solving time minus the rebuild times of every restart.

5.2.1 Confirming the "Compacting the Assignment Stack" Claim

We ran the Glucose 4.1 SAT solver [12] with various frequencies of restarting to show that indeed restarts do compact the assignment stack. For all experiments in this section, Glucose was run with the argument "-no-adapt" to prevent it from changing heuristics. For each instance in the SAT Competition 2017 main track, we ran Glucose 4.1 with a timeout of 3hrs of effective time on StarExec [82].

At every conflict, the assignment stack size is logged before backtracking occurs then the solver restarts after the conflict is analyzed (i.e., a uniform restart policy that restarts after every 1 conflict). We then ran the solver again on the same instance except the restart interval is doubled (i.e., a uniform restart policy that restarts after every 2 conflicts). We continue running the solver again and again, doubling the restart interval each time (i.e., a uniform restart policy that restarts after every 2^k conflicts) until the restart interval is so large that the solver never restarts before termination. For each instance, we construct a scatter plot, where the x-axis is the restart interval and the y-axis is the average assignment stack size for that restart policy on that instance, see Figure 5.2 for an example. We then compute the Spearman correlation between the two axes, a positive correlation denotes that smaller restart intervals correlate with smaller assignment stack size, that is evidence that frequent restarts compacts the assignment stack. We plot the Spearman correlations of all 350 instances in Figure 5.3. 91.7% of the instances have a positive correlation coefficient. In conclusion, our experiments support the claim by Hamadi et al. [45] "restarts policies are used to compact the assignment stack."

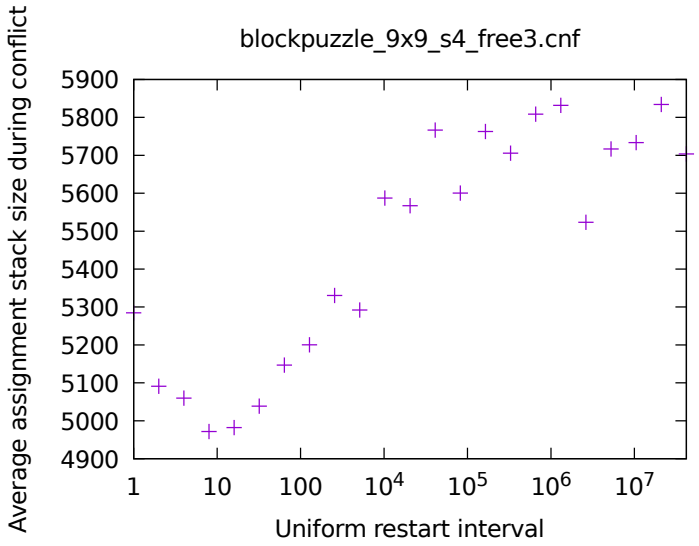

Figure 5.2: Scatter plot for a given instance showing increasing assignment stack size as the restarts become less frequent.

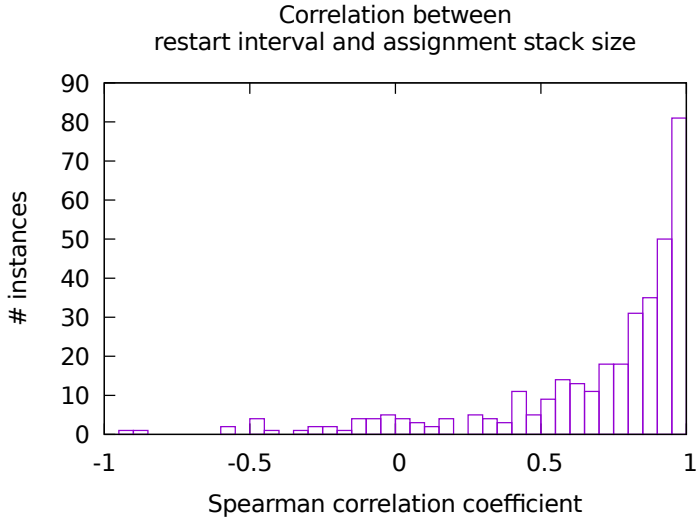

Figure 5.3: Histogram showing the distribution of Spearman correlations between the restart interval and the average assignment stack size for all 350 instances. The median correlation is 0.839.

It is important to note that this result is contingent on the branching heuristic implemented by the solver. If the branching heuristic is a static ordering, then the solver picks the decision variables in the same order after every restart and rebuilds the same assignment stack, hence the assignment stack does not get compacted. In our previous work [61], we showed that VSIDS-like branching heuristics "focus" on a small subset of logically related variables at any point in time. We believe a "focused" branching heuristic will see the compacting of assignment stack since a restart erases the assignment stack so a "focused" branching heuristic can reconstruct the assignment stack with only the subset of variables it is focused on. In the case of VSIDS and LRB, this subset of variables are the variables appearing in recent conflicts.

5.2.2 Learning Better Clauses

We hypothesize that compacting the assignment stack generally leads to better learnt clauses, and that this is one of the benefits of restarts in SAT solvers in practice. Note that the clause learning schemes construct the learnt clause from a subset of variables on the assignment stack. Hence, a smaller assignment stack should lead to a learnt clause with smaller LBD than otherwise. To show this experimentally, we repeat the previous experiment where we ran Glucose 4.1 with the uniform restart policy restarting every 2^k conflicts for various parameters of k. At each conflict, we log the assignment stack size before backtracking and the LBD of the newly learnt clause. For each instance, we draw a scatter plot, where the x-axis is the average assignment stack size and the y-axis is the average LBD of learnt clauses, see Figure 5.4. We compute the Spearman correlation between the two axes and plot these correlations in a histogram, see Figure 5.5. 73.1% of the instances have a positive correlation coefficient.

5.2.3 Solving Instances Faster

Although lower LBD is widely believed to be a sign of good quality clause, we empirically show that indeed lower LBD generally correlates with better effective time. This experiment is a repeat of the last two experiments, with the exception that the x-axis is the average learnt clause LBD and the y-axis is the effective time, see Figure 5.6 for an example. As usual, we compute the Spearman correlation between the two axes, discarding instances that timeout, and plot these correlations in a histogram, see Figure 5.5. 77.8% of the instances have a positive correlation coefficient. As expected, learning lower LBD clauses tend to improve solver performance.

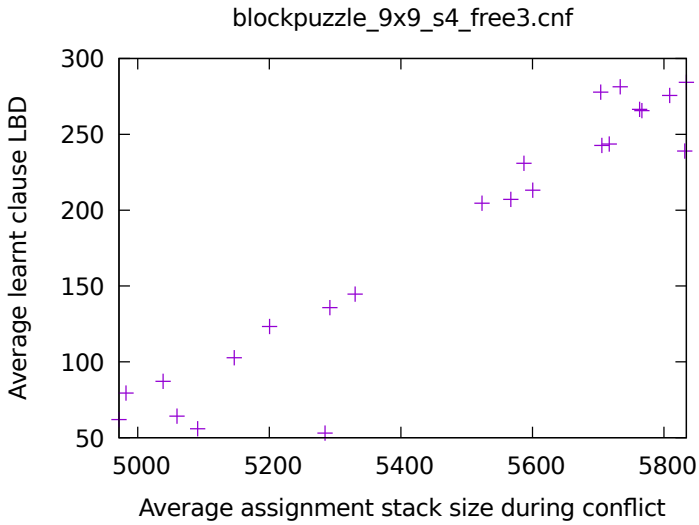

Figure 5.4: Scatter plot for a given instance showing increasing assignment stack size correlates with increasing LBD of learnt clauses.

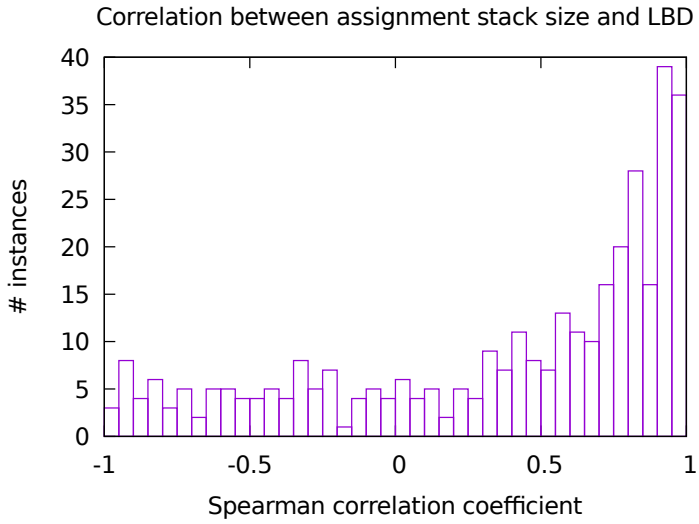

Figure 5.5: Histogram showing the distribution of Spearman correlations between the average assignment stack size and the average LBD of learnt clauses for all 350 instances. The median correlation is 0.607.

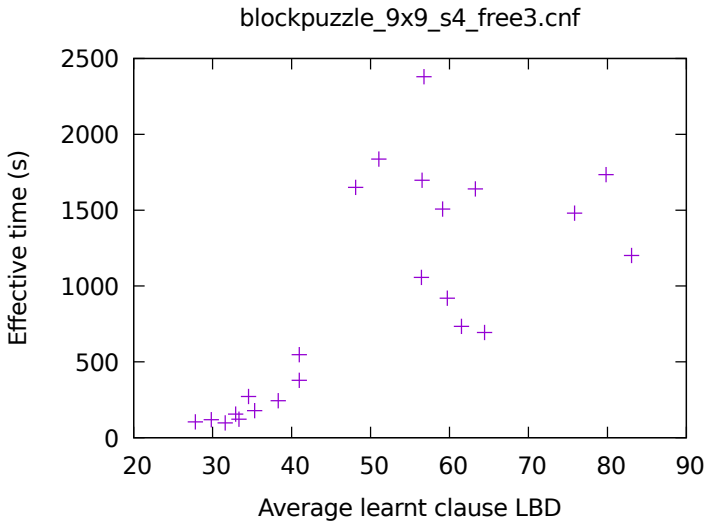

Figure 5.6: Scatter plot for a given instance showing increasing average learnt clause LBD correlates with increasing effective time.

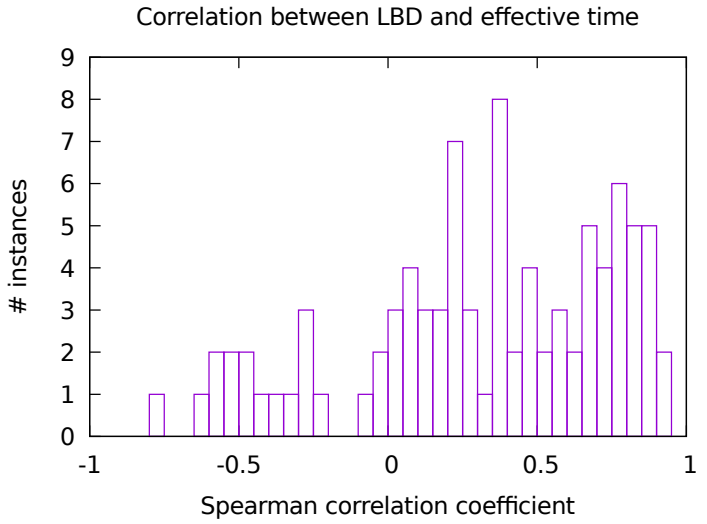

Figure 5.7: Histogram showing the distribution of Spearman correlations between the average learnt clause LBD and effective time for all 90 instances without timeouts. The median correlation is 0.366.

5.2.4 Clause Length

If the previous experiments replaced LBD with clause length, then the median Spearman correlation between the average assignment stack size and average learnt clause length is 0.822 and the median Spearman correlation between the average learnt clause length and effective time is 0.08. This lower correlation between clause length and effective time perhaps explains why LBD is generally a better metric than clause length.

5.2.5 Low LBD in Core Proof

We hypothesize that lower LBD clauses are preferable for unsatisfiable instances because they are more likely to be a core learnt clause, that is, a learnt clause that is actually used in the derivation of the final empty clause. We performed the following experiment to support our hypothesis. We ran Glucose with no clause deletion on all 350 instances of the SAT Competition 2017 main track with 5000 seconds timeout. We turned off clause deletion because the deletion policy in Glucose inherently biases towards low LBD clauses by deleting learnt clauses with higher LBDs. We used DRAT-trim [85] to extract the core proof from the output of Glucose, i.e, the subset of clauses used in the derivation of the empty clause. We then computed the ratio between the mean LBD of the core learnt clauses and the mean LBD of all the learnt clauses. Lastly we plotted the ratios in a histogram, see Figure 5.8. For the 57 instances for which core DRAT proofs were generated successfully, all but one instance has a ratio below 1. In other words, lower LBD clauses are more likely to be used in deriving the empty clause than clauses with higher LBD.

5.3 LBD Percentile

Given the LBD of a clause, it is unclear a priori how to label it as "good" or "bad". Some heuristics set a constant threshold and any LBDs above this threshold are considered bad. For example, the parallel solver Plingeling [15] considers learnt clauses with LBD greater 7 to be bad, and these clauses are not shared with the other solvers running concurrently. COMiniSatPS considers learnt clauses with LBD greater than 8 to be bad, and hence these clauses are readily deleted [73]. The state-of-the-art Glucose restart policy [11] on the other hand uses the mean LBD multiplied by a fixed constant as a threshold. The problem with using a fixed constant or the mean times a constant for thresholds is that we do not have a priori estimate of how many clauses exceed this threshold, and these thresholds seem

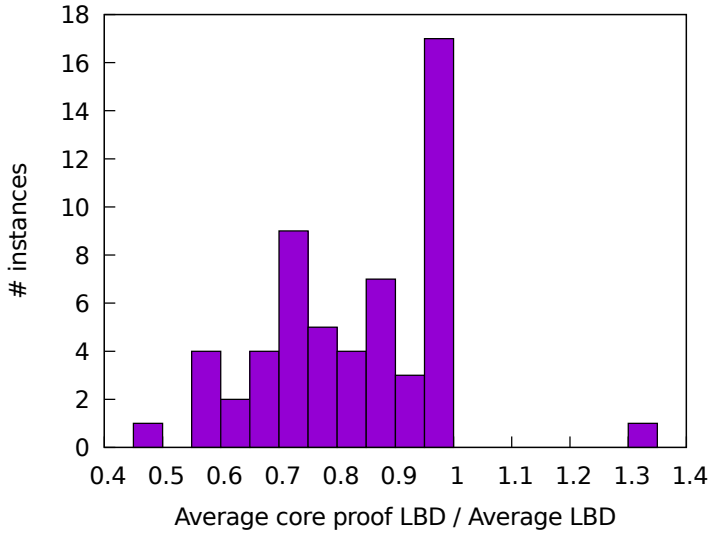

Figure 5.8: Histogram for the ratio between the mean LBD of the learnt clauses in the core proof and the mean LBD of all the learnt clauses for the 57 unsatisfiable instances DRAT-trim produced a core proof.

arbitrary. Using arbitrary thresholds makes it harder to reason about solver heuristics, and in this context, the efficacy of restart policies.

For the above reasons, we propose that it is more appropriate to use dynamic threshold. The threshold is dynamically computed by collecting empirical data from the CDCL SAT solver during its run for a given input. We chose the threshold to be the 99.9^{th} percentile of LBDs of the learnt clauses seen during the run so far. Before we estimate whether an LBD is in the 99.9^{th} percentile, the first step is to analyze the distribution of LBDs seen in practice. In this experiment, the Glucose solver was run on all 350 instances in SAT Competition 2017 main track for 30 minutes and the LBDs of all the learnt clauses were recorded. Figure 5.9 shows the histogram of LBDs for 4 representative instances. As can be seen from the distributions of these representative instances, either their LBD distribution is close to normal or a right-skewed one.

Even though the right-skew distribution is not normal, the high percentiles can still be approximated by the normal distribution since the right tail is close to the normal curve. We conducted the following experiment to support this claim. For each instance, we computed the mean and variance of the LBD distribution to draw a normal distribution with the same mean and variance. We used the normal distribution to predict the LBD x

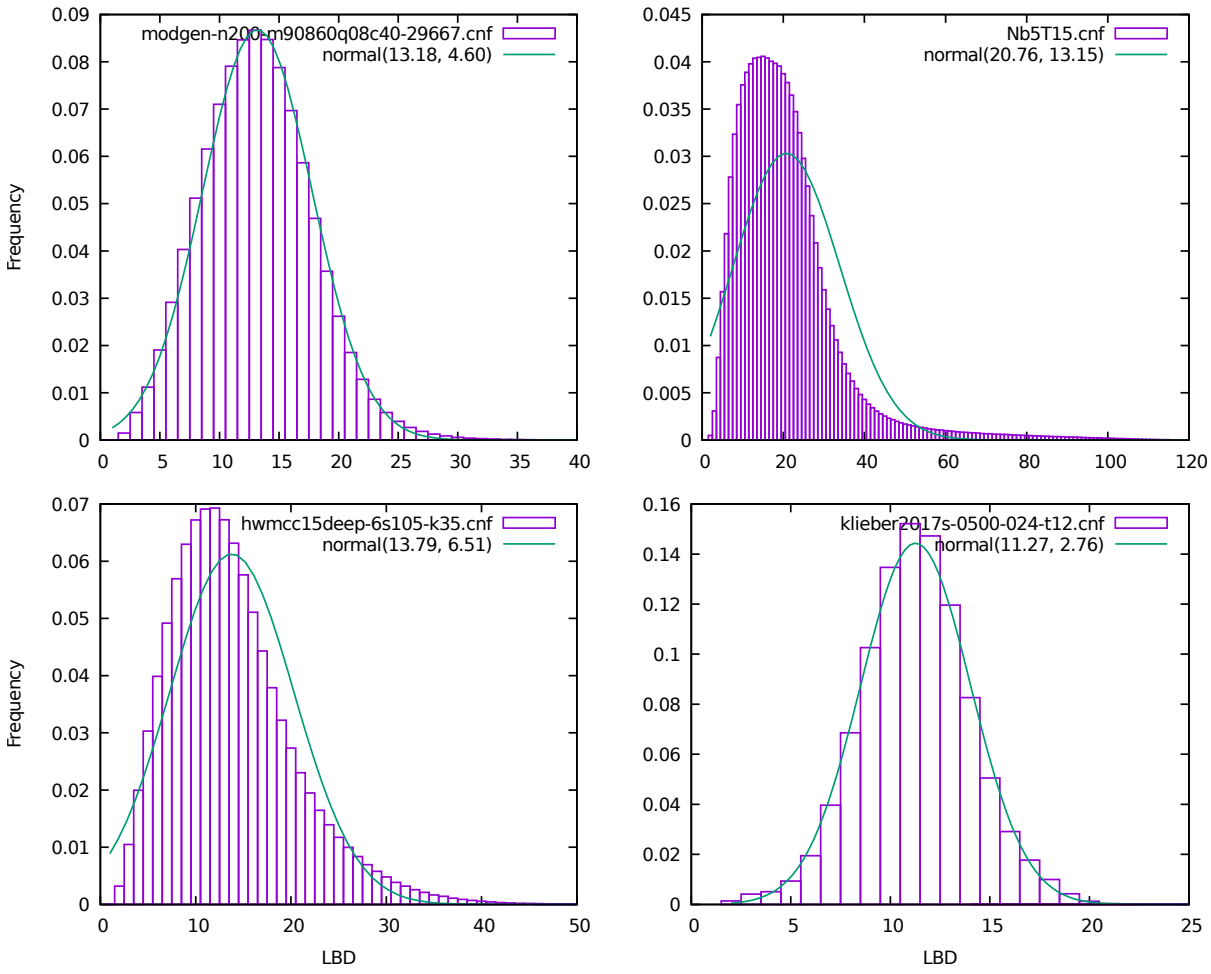

Figure 5.9: Histogram of LBDs of 4 instances. A normal distribution with the same mean and variance is overlaid on top for comparison.

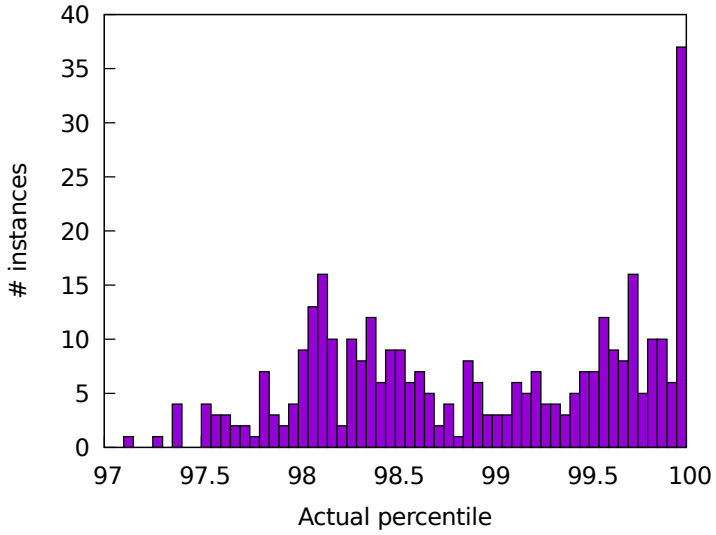

Figure 5.10: Histogram of the actual percentiles of the LBD predicted to be the 99.9^{th} percentile using a normal distribution.

at the 99.9^{th} percentile. We then checked the recorded LBD distribution to see the actual percentile of x. Figure 5.10 is a histogram of all the actual percentiles. Even in the worst case, the predicted 99.9^{th} percentile turned out to be the 97.1^{th} percentile. Hence for this benchmark the prediction of the 99.9^{th} percentile using the normal distribution has an error of less than 3 percentiles. Additionally, only 6 of the 350 instances predicted an LBD that was in the 100^{th} percentile and all 6 of these instances solved in less than 130 conflicts hence the prediction was made with very little data.

These figures were created by analyzing the LBD distribution at the end of a 30 minute run of Glucose, and we note the results are similar before the 30 minutes are up. Hence the 99.9^{th} percentile of LBDs can be approximated as the 99.9^{th} percentile of $norm(\mu, \sigma^2)$. The mean μ and variance σ^2 are estimated by the sample mean and sample variance of all the LBDs seen thus far, which is computed incrementally so the computational overhead is low. The 99.9^{th} percentile of the normal distribution maps to the z-score of 3.08, that is, an LBD is estimated to be in the 99.9^{th} percentile if it is greater than $\mu + 3.08 \times \sigma$.

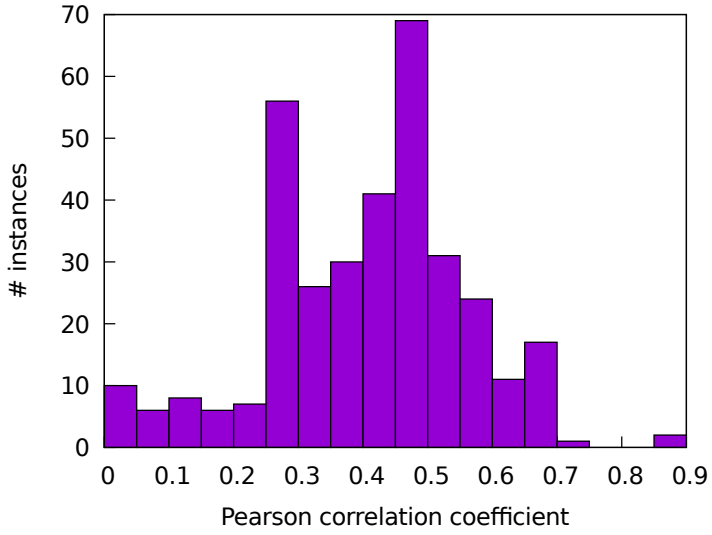

Figure 5.11: Histogram of the Pearson correlation between the "previous" and "next" LBD for the instances in the SAT Competition 2017 main track benchmark.

5.4 LBD of Next Clause

Since at any point during the run of a solver, the LBD of the "next learnt" clause is unknown, we propose the use of machine learning to predict that LBD instead. This requires finding good features that correlate with the next LBD. We hypothesize that LBDs of recent past learnt clauses correlate with the LBD of the next learnt clause.

In this experiment, Glucose was run on all 350 instances of the 2017 Competition main track and the LBDs of all the learnt clauses were recorded. Let n be the number of LBDs recorded for an instance. A table with two columns of length $n-1$ are created. For each row i in this two column table, the first column contains the LBD of the i^{th} conflict and the second column contains the LBD of the $(i+1)^{th}$ conflict. Intuitively, after the solver finishes resolving the i^{th} conflict, the i^{th} learnt clause is the "previous" learnt clause represented by the first column. Correspondingly, the "next" learnt clause is the $(i+1)^{th}$ learnt clause represented by the second column. For each instance that took more than 100 conflicts to solve, we computed the Pearson correlation between the first and second column and plot all these correlations in a histogram, see Figure 5.11.

Our results show that the "previous LBD" is correlated with the "next LBD" which supports the idea that recent LBDs are good features to predict the next LBD via machine

learning. In addition, all the correlations are positive, meaning that if the previous LBD is high (resp. low) then the next LBD is expected to be high (resp. low). Perhaps this explains why the Glucose restart policy [11] is effective. Additionally, we note that for the average instance, the LBD of the learnt clause after a restart is smaller than the LBD of the learnt clause right before that restart 74% of the time, showing the effect of restarts on LBD.

We propose learning the function $f(l_{-1}, l_{-2}, l_{-3}, l_{-1} \times l_{-2}, l_{-1} \times l_{-3}, l_{-2} \times l_{-3}) = l_{next}$ where l_{-i} is the LBD of the learnt clause from i conflicts ago and $l_{-i} \times l_{-j}$ are products of previous LBDs to incorporate their feature interaction, and l_{next} is the LBD of the next clause. This function is approximated using linear regression where θ_i are coefficients to be trained by the machine learning algorithm:

$\tilde{f}(l_{-1}, l_{-2}, l_{-3}, l_{-1} \times l_{-2}, l_{-1} \times l_{-3}, l_{-2} \times l_{-3}) = \theta_0 + \theta_1 \times l_{-1} + \theta_2 \times l_{-2} + \theta_3 \times l_{-3} + \theta_4 \times l_{-1} \times l_{-2} + \theta_5 \times l_{-1} \times l_{-3} + \theta_6 \times l_{-2} \times l_{-3}$

Since LBDs are streamed in as conflicts occur, an online algorithm that can incrementally adjust the θ_i coefficients cheaply is required. We use the state-of-the-art Adam algorithm [55] from machine learning literature because it scales well with the number of dimensions, is computationally efficient, and converges quickly for many problems. The Adam algorithm is in the family of stochastic gradient descent algorithms that adjusts the coefficients to minimize the squared error, where the error is the difference between the linear function's prediction and the actual next LBD. The algorithm computes the gradient of the squared error function and adjusts the coefficients in the opposite direction of the gradient to minimize the squared error function. For the parameters of Adam, we use the values recommended by the original authors [55].

The coefficients θ_i are all initialized to 0 at the start of the search. Whenever a new clause is learnt, one iteration of Adam is applied with the LBDs of the three previous learnt clauses and their pairwise products as features and the LBD of the new clause as the target. The coefficients θ_i are adjusted in the process. When BCP reaches a fixed point without a conflict, the function $\tilde{f}$ is queried with the current set of coefficients θ_i to predict the LBD of the next clause. If the prediction exceeds the sample mean plus 3.08 standard deviations (i.e., approximately the 99.9^{th} percentile), a restart is triggered.

The new restart policy, called *machine learning-based restart* (MLR) policy, is shown in Algorithm 10. Since the mean, variance, and coefficients are computed incrementally, MLR has a very low computational overhead.

Algorithm 10 Pseudocode for the new restart policy MLR.

function INITIALIZE ▷ Called once at the start of search.
  $\alpha \leftarrow 0.001, \epsilon \leftarrow 0.00000001, \beta_1 \leftarrow 0.9, \beta_2 \leftarrow 0.999$ ▷ Adam parameters.
  $conflicts \leftarrow 0, conflictsSinceLastRestart \leftarrow 0$
  $t \leftarrow 0$ ▷ Number of training examples.
  $prevLbd_3 \leftarrow 0, prevLbd_2 \leftarrow 0, prevLbd_1 \leftarrow 0$ ▷ LBD of clause learnt 3/2/1 conflicts ago.
  $\mu \leftarrow 0, m2 \leftarrow 0$ ▷ For computing sample mean and variance of LBDs seen.
  for v in $0..|FeatureVector()| - 1$ **do** ▷ Initialize θ, m, v to be vectors of zeros.
    $\theta_i \leftarrow 0, m_i \leftarrow 0, v_i \leftarrow 0$ ▷ Coefficients of linear function and Adam internals.
  end for
end function

function FEATUREVECTOR
  return $[1, prevLbd_1, prevLbd_2, prevLbd_3, prevLbd_1 \times prevLbd_2, prevLbd_1 \times prevLbd_3, prevLbd_2 \times prevLbd_3]$
end function

function AFTERCONFLICT(LearntClause) ▷ Update the coefficients θ using Adam.
  $conflicts \leftarrow conflicts + 1, conflictsSinceLastRestart \leftarrow conflictsSinceLastRestart + 1$
  $nextLbd \leftarrow LBD(LearntClause)$
  $\delta \leftarrow nextLbd - \mu, \mu \leftarrow \mu + \delta/conflicts, \Delta \leftarrow nextLbd - \mu, m2 \leftarrow m2 + \delta \times \Delta$
  if $conflicts > 3$ **then** ▷ Apply one iteration of Adam.
    $t \leftarrow t + 1$
    $features \leftarrow FeatureVector()$
    $predict \leftarrow \theta \cdot features$
    $error \leftarrow predict - nextLbd$
    $g \leftarrow error \times features$
    $m \leftarrow \beta_1 \times m + (1 - \beta_1) \times g, v \leftarrow \beta_2 \times v + (1 - \beta_2) \times g \times g$
    $\hat{m} \leftarrow m/(1 - \beta_1^t), \hat{v} \leftarrow v/(1 - \beta_2^t)$
    $\theta \leftarrow \theta - \alpha \times \hat{m}/(\sqrt{\hat{v}} + \epsilon)$
  end if
  $prevLbd_3 \leftarrow prevLbd_2, prevLbd_2 \leftarrow prevLbd_1, prevLbd_1 \leftarrow nextLbd$
end function

function AFTERBCP(IsConflict)
  if $\neg IsConflict \wedge conflicts > 3 \wedge conflictsSinceLastRestart > 0$ **then**
    $\sigma \leftarrow \sqrt{m2/(conflicts - 1)}$
    if $\theta \cdot FeatureVector() > \mu + 3.08\sigma$ **then** ▷ Estimate if next LBD in 99.9th percentile.
      $conflictsSinceLastRestart \leftarrow 0, Restart()$
    end if
  end if
end function

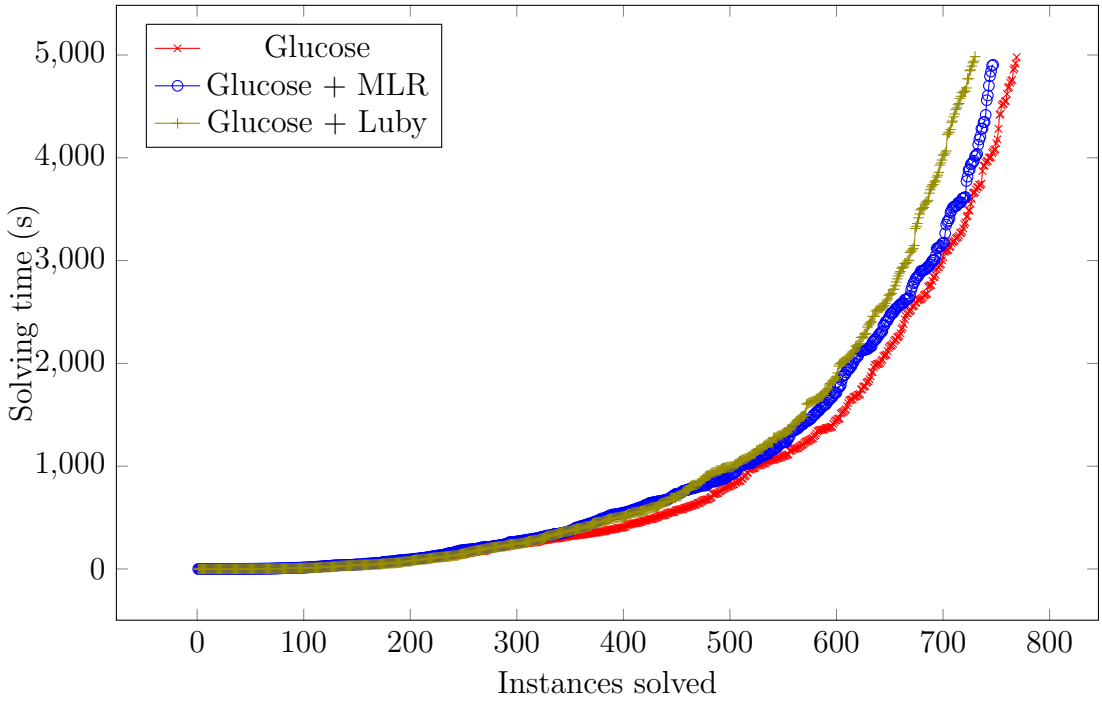

Figure 5.12: Cactus plot of two state-of-the-art restart policies and MLR on the entire benchmark with duplicate instances removed.

5.5 Experimental Evaluation

To test how MLR performs, we conducted an experimental evaluation to see how Glucose performs with various restart policies. Two state-of-the-art restart policies are used for comparison with MLR: Glucose (named after the solver itself) [11] and Luby [63]. The benchmark consists of all instances in the application and hard combinatorial tracks from the SAT Competition 2014 to 2017 totaling 1411 unique instances. The Glucose solver with various restart policies were run over the benchmark on StarExec. For each instance, the solver was given 5000 seconds of CPU time and 8GB of RAM. The results of the experiment are shown in Figure 5.12. The source code of MLR and further analysis of the experimental results are available on our website [5].

The results show that MLR is in between the two state-of-the-art policies of Glucose restart and Luby restart. For this large benchmark, MLR solves 19 instances more than Luby and 20 instances fewer than Glucose. Additionally, the learnt coefficients in MLR $\sigma_1, \sigma_2, \sigma_3$ corresponding to the coefficients of the features representing recent past LBDs

are nonnegative 91% of the time at the end of the run. This reinforces the notion that previous LBDs are positively correlated with the next LBD.

5.6 Related Work

Theorists have conjectured that restarts give the solver more power in a proof-complexity sense than a solver without restarts. A CDCL solver with asserting clause learning scheme can polynomially simulate general resolution [75] with nondeterministic branching and restarts. It was independently shown that a CDCL solver with sufficiently random branching and restarts can simulate bounded-width resolution [9]. It remains an open question whether these results hold if the solvers does not restart. This question has remained stubbornly open for over two decades now. We refer the reader to the excellent articles by Buss et al. on attempts at understanding the power of restarts via proof-complexity theory [23, 19].

Restart policies come in two flavors: static and dynamic. Static restart policies predetermine when to restart before the search begins. The state-of-the-art for static is the Luby [63] restart heuristic which is theoretically proven to be an optimal universal restart policy for Las Vegas algorithms. Dynamic restart policies determine when to restart on-the-fly during the run of the solver, typically by analyzing solver statistics. The state-of-the-art for dynamic is the restart policy proposed by Glucose [11] that keeps a short-term and a long-term average of LBDs. The short-term is the average of the last 50 LBDs and the long-term is the average of all the LBDs encountered since the start of the search. If the short-term exceeds the long-term by a constant factor then a restart is triggered. Hence the Glucose policy triggers a restart when the recent LBDs are high on average whereas MLR restarts when the predicted LBD of the next clause is high. Biere et al. [16] propose a variation of the Glucose restart where an exponential moving average is used to compute the short-term and long-term averages. Haim and Walsh [44] introduced a machine learning-based technique to select a restart policy from a portfolio after 2100 conflicts. The MABR policy [70] uses multi-armed bandits to minimize average LBD by dynamically switching between a portfolio of policies. Our use of machine learning differs from these previous methods in that machine learning is part of the restart policy itself, rather than using machine learning as a meta-heuristic to select between a fixed set of restart policies.

Chapter 6

Conclusion

Heuristic design for CDCL SAT solvers is very challenging due to the large design space and our poor understanding of what makes CDCL SAT solvers tick. On the other hand, it is also very rewarding due to their prominent role in the overall effectiveness of the solver. This thesis provides set of pragmatic principles in heuristic design that proved to be successful in the branching heuristic and restart policy space.

1. Find a simple metric for the given heuristic space that correlates with solving time. For branching heuristics, we used the global learning rate (GLR) as the metric. For restart policies, the metric we used is the literal block distance (LBD). In our experience with branching and restarts, we defined the metric with respect to some aspect of clause learning since CDCL SAT solvers are proof systems that attempt to construct relatively small proofs. In the case of branching, we optimized for the quantity of learning (GLR) and whereas in restarts we optimized for the quality (LBD). We believe that a fruitful future direction is to incorporate multiple metrics that encompass a wider perspective of SAT performance and invent new heuristics that simultaneously optimize all the metrics in an multi-objective optimization fashion.

2. Observe what happens when said metric is optimized when overhead is discounted. We designed the greedy GLR branching (GGB) heuristic to show that maximizing GLR improves performance if the huge overhead is ignored. Although this heuristic is grossly impractical, its purpose is a proof-of-concept to demonstrate the effects of maximizing GLR. It serves as a gold standard for subsequent heuristics to emulate. This provides a good sanity check that the metric makes sense in practice.

3. Exploit machine learning to perform the optimization cheaply. On the branching heuristic space, we used machine learning to directly approximate GGB with the stochastic gradient descent branching (SGDB) heuristic. On the restart policy space, we used machine learning to predict upcoming learnt clauses with poor LBD. In either case, we took advantage of the data generated from conflict analysis during the run of the solver to learn the respective functions. This gives rise to the online machine learning-based heuristics, rather than applying machine learning outside of the SAT solver like in the portfolio solver approach. This avoids a common overfitting problem when training offline over a training set that is distinct from the instance being solved. Even instances that look similar superficially can have very different runtime characteristics, and offline training has a very difficult time generalizing between instances.

SAT solvers naturally have many difficult choices to make. Which variable to branch on? When to restart? Which clause to learn and which clause to delete? With the explosive rise of machine learning, it is becoming abundantly clear these choices should be learnt rather than the old-fashioned way of handcoding a strategy. With the success in the branching heuristic and restart policy domain, we are confident in the wisdom of using machine learning techniques as a basis for designing heuristics in CDCL SAT solvers.

Our machine learning work partitions the solver nicely into two components that is metaphorically similar to human reasoning. The first component is a deductive reasoning *teacher*. This includes BCP and conflict analysis that applies the resolution proof rule to deduce lemmas. However, deductive learning causes a blow up since there is an exponential number of lemmas. The second component is an inductive reasoning *student*. This includes the heuristics that apply machine learning to learn from data which part of the search space is most productive. However, the conclusions derived from inductive reasoning may be unsound, but thankfully the deductive reasoning blocks any mistakes and returns feedback to the heuristics to fine-tune their inductive reasoning. Viewing the SAT solver as a student-teacher loop makes it clear the similarities between CDCL and other verification and learning algorithms.

Reinforcement learning: In *reinforcement learning* [83], the student is the agent learning to perform the best actions online from feedback it receives from the teacher environment. The student estimates the best actions at each timestep and the teacher corrects the student by using rewards to steer the student towards optimal actions. The process repeats until the student converges on the optimal action. The student-teacher similarity between CDCL and reinforcement learning led us to developing the LRB algorithm.

Query learning: In *query learning* [6], the student is trying to learn a target language by hypothesizing a language to the teacher who then returns a counterexample as to why the hypothesized language is different than the target language. The teacher guides the student's hypothesis through this counterexample feedback mechanism. The process repeats until until the student's hypothesis is exactly the target language.

CEGAR: In *counterexample-guided abstraction refinement* (CEGAR) [27], the student synthesizes a model that ideally adheres to the specification. If the model is spurious, the teacher explains the error to the student who uses the explanation to refine its model. The process repeats until the student finds a correct model.

As we can see, the student-teacher model is quite general in the space of learning and verification. The student synthesizes potential answers (i.e., partial assignments, actions, languages, models) for the teacher to critique. The teacher's feedback (i.e., learnt clauses, rewards, counterexamples) is used by the student to refine its understanding. The student and teacher form a continuous feedback loop until the student converges on the correct solution. As we can see, these types of algorithms are very powerful in their respective domains. In this thesis, we introduced the idea of using machine learning to implement the student so that it learns to synthesize the correct partial assignments over time. We hope that our use of a machine learning student can inspire other incarnations of the student-teacher model to take advantage of machine learning as well.

References

[1] http://www.satcompetition.org/.

[2] https://www.sharcnet.ca.

[3] https://sites.google.com/a/gsd.uwaterloo.ca/maplesat/.

[4] https://sites.google.com/a/gsd.uwaterloo.ca/maplesat/sgd.

[5] https://sites.google.com/a/gsd.uwaterloo.ca/maplesat/mlr.

[6] Dana Angluin. Queries and Concept Learning. *Machine Learning*, 2(4):319–342, April 1988.

[7] Carlos Ansótegui, Jesús Giráldez-Cru, and Jordi Levy. The Community Structure of SAT Formulas. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing – SAT 2012*, pages 410–423, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[8] Carlos Ansótegui, Jesús Giráldez-Cru, Jordi Levy, and Laurent Simon. Using Community Structure to Detect Relevant Learnt Clauses. In Marijn Heule and Sean Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015*, pages 238–254, Cham, 2015. Springer International Publishing.

[9] Albert Atserias, Johannes Klaus Fichte, and Marc Thurley. Clause-learning algorithms with many restarts and bounded-width resolution. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, pages 114–127, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[10] Gilles Audemard and Laurent Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, IJCAI'09, pages 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[11] Gilles Audemard and Laurent Simon. *Refining Restarts Strategies for SAT and UNSAT*, pages 118–126. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[12] Gilles Audemard and Laurent Simon. Glucose 2.3 in the SAT 2013 Competition. In *Proceedings of SAT Competition 2013*, pages 42–43, 2013.

[13] Armin Biere. Adaptive restart strategies for conflict driven SAT solvers. In *Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing*, SAT'08, pages 28–33, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] Armin Biere. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. *FMV Report Series Technical Report*, 10(1), 2010.

[15] Armin Biere. Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013. In *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, pages 51–52, 2013.

[16] Armin Biere and Andreas Fröhlich. *Evaluating CDCL Variable Scoring Schemes*, pages 405–422. Springer International Publishing, Cham, 2015.

[17] Armin Biere and Andreas Fröhlich. *Theory and Applications of Satisfiability Testing – SAT 2015: 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, chapter Evaluating CDCL Variable Scoring Schemes, pages 405–422. Springer International Publishing, Cham, 2015.

[18] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008.

[19] Maria Luisa Bonet, Sam Buss, and Jan Johannsen. Improved Separations of Regular Resolution from Clause Learning Proof Systems. *Journal of Artificial Intelligence Research*, 49:669–703, 2014.

[20] Léon Bottou. On-line Learning in Neural Networks. chapter On-line Learning and Stochastic Approximations, pages 9–42. Cambridge University Press, New York, NY, USA, 1998.

[21] Aaron R. Bradley. SAT-based Model Checking Without Unrolling. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'11, pages 70–87, Berlin, Heidelberg, 2011. Springer-Verlag.

[22] Robert G Brown. Exponential Smoothing for Predicting Demand. In *Operations Research*, volume 5, pages 145–145, 1957.

[23] Samuel R. Buss and Leszek Aleksander Kolodziejczyk. Small Stone in Pool. *Logical Methods in Computer Science*, 10(2), 2014.

[24] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[25] Elsa Carvalho and João P. Marques Silva. 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.

[26] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[27] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification*, CAV '00, pages 154–169, London, UK, UK, 2000. Springer-Verlag.

[28] Aaron Clauset, M. E. J. Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical Review E*, 70:066111, Dec 2004.

[29] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[30] D. R. Cox. The Regression Analysis of Binary Sequences. *Journal of the Royal Statistical Society. Series B (Methodological)*, 20(2):215–242, 1958.

[31] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[32] Niklas Eén and Niklas Sörensson. *Theory and Applications of Satisfiability Testing: 6th International Conference, SAT 2003, Santa Margherita Ligure, Italy, May 5-8, 2003, Selected Revised Papers*, chapter An Extensible SAT-solver, pages 502–518. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[33] Ido Erev and Alvin E Roth. Predicting How People Play Games: Reinforcement Learning in Experimental Games with Unique, Mixed Strategy Equilibria. *American Economic Review*, 88(4):848–881, 1998.

[34] Katherine Faust. Centrality in affiliation networks. *Social Networks*, 19(2):157 – 191, 1997.

[35] R. A. Fisher. Frequency Distribution of the Values of the Correlation Coefficient in Samples from an Indefinitely Large Population. *Biometrika*, 10(4):507–521, 1915.

[36] Linton C. Freeman. Centrality in Social Networks Conceptual Clarification. *Social Networks*, 1(3):215 – 239, 1978.

[37] Andreas Fröhlich, Armin Biere, Christoph Wintersteiger, and Youssef Hamadi. Stochastic Local Search for Satisfiability Modulo Theories. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pages 1136–1143. AAAI Press, 2015.

[38] Roman Gershman and Ofer Strichman. *Hardware and Software, Verification and Testing: First International Haifa Verification Conference, Haifa, Israel, November 13-16, 2005, Revised Selected Papers*, chapter HaifaSat: A New Robust SAT Solver, pages 76–89. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[39] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002.

[40] Eugene Goldberg and Yakov Novikov. BerkMin: A Fast and Robust Sat-solver. *Discrete Appl. Math.*, 155(12):1549–1561, June 2007.

[41] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[42] Carla P. Gomes, Bart Selman, Nuno Crato, and Henry Kautz. Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems. *Journal of Automated Reasoning*, 24(1-2):67–100, February 2000.

[43] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting Combinatorial Search Through Randomization. In *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, AAAI '98/IAAI '98, pages 431–437, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[44] Shai Haim and Toby Walsh. Restart Strategy Selection Using Machine Learning Techniques. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, pages 312–325, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[45] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. ManySAT: a Parallel SAT solver. *Journal on Satisfiability*, 6:245–262, 2008.

[46] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing – SAT 2016*, pages 228–245, Cham, 2016. Springer International Publishing.

[47] Edward A. Hirsch and Arist Kojevnikov. UnitWalk: A new SAT solver that uses local search guided by unit clause elimination. *Annals of Mathematics and Artificial Intelligence*, 43(1):91–111, Jan 2005.

[48] Ruoyun Huang, Yixin Chen, and Weixiong Zhang. SAS+ Planning As Satisfiability. *Journal of Artificial Intelligence Research*, 43(1):293–328, January 2012.

[49] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential Model-Based Optimization for General Algorithm Configuration. In Carlos A. Coello Coello, editor, *Learning and Intelligent Optimization*, pages 507–523, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[50] Markus Iser, Mana Taghdiri, and Carsten Sinz. Optimizing MiniSAT variable orderings for the relational model finder Kodkod. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing*, SAT'12, pages 483–484, Berlin, Heidelberg, 2012. Springer-Verlag.

[51] Robert G. Jeroslow and Jinchang Wang. Solving Propositional Satisfiability Problems. *Annals of Mathematics and Artificial Intelligence*, 1(1-4):167–187, September 1990.

[52] Hadi Katebi, Karem A. Sakallah, and João P. Marques-Silva. Empirical Study of the Anatomy of Modern Sat Solvers. In *Proceedings of the 14th International Conference on Theory and Application of Satisfiability Testing*, SAT'11, pages 343–356, Berlin, Heidelberg, 2011. Springer-Verlag.

[53] George Katsirelos and Laurent Simon. Eigenvector Centrality in Industrial SAT Instances. In Michela Milano, editor, *Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, pages 348–356. Springer Berlin Heidelberg, 2012.

[54] Henry Kautz and Bart Selman. Planning As Satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence*, ECAI '92, pages 359–363, New York, NY, USA, 1992. John Wiley & Sons, Inc.

[55] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *CoRR*, abs/1412.6980, 2014.

[56] Boris Konev and Alexei Lisitsa. A sat attack on the erdős discrepancy conjecture. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014*, pages 219–226, Cham, 2014. Springer International Publishing.

[57] Lars Kotthoff, Ian P. Gent, and Ian Miguel. An Evaluation of Machine Learning in Algorithm Selection for Search Problems. *AI Communications*, 25(3):257–270, August 2012.

[58] Michail G Lagoudakis and Michael L Littman. Learning to Select Branching Rules in the DPLL Procedure for Satisfiability. *Electronic Notes in Discrete Mathematics*, 9:344–359, 2001.

[59] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Exponential Recency Weighted Average Branching Heuristic for SAT Solvers. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 3434–3440. AAAI Press, 2016.

[60] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning Rate Based Branching Heuristic for SAT Solvers. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing – SAT 2016*, pages 123–140, Cham, 2016. Springer International Publishing.

[61] Jia Hui Liang, Vijay Ganesh, Ed Zulkoski, Atulan Zaman, and Krzysztof Czarnecki. Understanding VSIDS Branching Heuristics inConflict-Driven Clause-Learning SAT Solvers. In Nir Piterman, editor, *Hardware and Software: Verification and Testing*, pages 225–241, Cham, 2015. Springer International Publishing.

[62] Manuel Loth, Michèle Sebag, Youssef Hamadi, and Marc Schoenauer. *Principles and Practice of Constraint Programming: 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, chapter Bandit-Based Search for Constraint Programming, pages 464–480. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[63] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal Speedup of Las Vegas Algorithms. *Information Processing Letters*, 47(4):173–180, September 1993.

[64] Manuel Lpez-Ibez, Jrmie Dubois-Lacoste, Leslie Prez Cceres, Mauro Birattari, and Thomas Sttzle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43 – 58, 2016.

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

[66] João P Marques-Silva and Karem A. Sakallah. GRASP-A New Search Algorithm for Satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[67] Matthew W. Moskewicz, Conor F. Madigan, and Sharad Malik. Method and system for efficient implementation of boolean satisfiability, August 26 2008. US Patent 7,418,369.

[68] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[69] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[70] Saeed Nejati, Jia Hui Liang, Catherine Gebotys, Krzysztof Czarnecki, and Vijay Ganesh. Adaptive Restart and CEGAR-Based Solver for Inverting Cryptographic Hash Functions. In Andrei Paskevich and Thomas Wies, editors, *Verified Software. Theories, Tools, and Experiments*, pages 120–131, Cham, 2017. Springer International Publishing.

[71] M. E. J. Newman. Analysis of weighted networks. *Physical Review E*, 70:056131, Nov 2004.

[72] Zack Newsham, Vijay Ganesh, Sebastian Fischmeister, Gilles Audemard, and Laurent Simon. Impact of Community Structure on SAT Solver Performance. In Carsten Sinz

and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014*, pages 252–268, Cham, 2014. Springer International Publishing.

[73] Chanseok Oh. *Improving SAT Solvers by Exploiting Empirical Characteristics of CDCL*. PhD thesis, New York University, 2016.

[74] Knot Pipatsrisawat and Adnan Darwiche. A Lightweight Component Caching Scheme for Satisfiability Solvers. In *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing*, SAT'07, pages 294–299, Berlin, Heidelberg, 2007. Springer-Verlag.

[75] Knot Pipatsrisawat and Adnan Darwiche. *On the Power of Clause-Learning SAT Solvers with Restarts*, pages 654–668. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[76] Knot Pipatsrisawat and Adnan Darwiche. On Modern Clause-Learning Satisfiability Solvers. *Journal of Automated Reasoning*, 44(3):277–301, March 2010.

[77] Antonio Ramos, Peter van der Tak, and Marijn J. H. Heule. Between Restarts and Backjumps. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011*, pages 216–229, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[78] Lawrence Ryan. Efficient Algorithms for Clause-Learning SAT Solvers. Master's thesis, Simon Fraser University, 2004.

[79] Lorenza Saitta and Michele Sebag. *Phase Transitions in Machine Learning*, pages 767–773. Springer US, Boston, MA, 2010.

[80] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT Solvers to Cryptographic Problems. In *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, pages 244–257, 2009.

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

[82] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. *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*, chapter StarExec: A Cross-Community Infrastructure for Logic Solving, pages 367–373. Springer International Publishing, Cham, 2014.

[83] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*, volume 1. MIT press Cambridge, 1998.

[84] G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983.

[85] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt. DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014*, pages 422–429, Cham, 2014. Springer International Publishing.

[86] Ryan Williams, Carla P. Gomes, and Bart Selman. Backdoors to Typical Case Complexity. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, IJCAI'03, pages 1173–1178, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.

[87] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *Journal of Artificial Intelligence Research*, 32(1):565–606, June 2008.

[88] Eldad Yechiam and Jerome R. Busemeyer. Comparison of basic assumptions embedded in learning models for experience-based decision making. *Psychonomic Bulletin & Review*, 12(3):387–402, June 2005.

[89] Wangsheng Zhang, Gang Pan, Zhaohui Wu, and Shijian Li. Online community detection for large complex networks. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, pages 1903–1909. AAAI Press, 2013.

[90] Edward Zulkoski, Curtis Bright, Albert Heinle, Ilias Kotsireas, Krzysztof Czarnecki, and Vijay Ganesh. Combining sat solvers with computer algebra systems to verify combinatorial conjectures. *Journal of Automated Reasoning*, 58(3):313–339, Mar 2017.