Understanding and Improving SAT Solvers via Proof Complexity and Reinforcement Learning

by

Chunxiao Li

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, 2023

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:	Laurent Simon Professor, Bordeaux - INP, Laboratoire Bordelais de Recherche en Informatique (LABRI)
Supervisor(s):	Vijay Ganesh Associate 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 Krzysztof Czarnecki Professor, Dept. of Electrical and Computer Engineering, University of Waterloo
Internal-External Member:	Pascal Poupart Professor, David R. Cheriton School of Computer Science, University of Waterloo

Author's Declaration

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

Despite the fact that the Boolean satisfiability (SAT) problem is NP-complete and believed to be intractable, SAT solvers are routinely used by practitioners to solve hard problems in wide variety of fields such as software engineering, formal methods, security, and AI. This gap between theory and practice has motivated an entire line of research whose primary goals are twofold: first, to develop a better theoretical framework aimed at accurately capturing solver behavior and thus prove tighter complexity bounds; and second, to further experimentally verify the soundness of the theory thus developed via rigorous empirical analysis and design theory-inspired techniques to improve solver performance. This interplay between theory and practice is at the heart of the work presented here.

More precisely, this thesis contains a collection of results which attempt to resolve the above-described discrepancy between theory and practice. The first two sets of results are centered around the restart problem. Restarts are classes of methods which aim at erasing part of the progress a solver may have made at run time, in order to help solvers escape from the "bad parts" of the search space. We provide a detailed theoretical analysis of the power of restarts used in modern Conflict-Driven Clause Learning (CDCL) SAT solvers, where we prove a series of equivalence and separation results for various configurations of solvers with and without restarts. From the intuition developed via this theoretical analysis, we design and implement a machine learning based reset policy, where resets are variants of restarts that erase activity scores in addition to the parts of the solver state erased by restarts. We perform extensive experimental work to show that our reset policy outperforms both baseline and state-of-the-art solvers over a class of cryptographic instances derived from bitcoin mining problems.

In a different direction, we propose the concept of hierarchical community structure (HCS) for Boolean formulas. We first theoretically show that formulas with "good" HCS parameter values have short CDCL proofs. Then we construct an Empirical Hardness Model using the HCS parameters. These HCS parameters exhibit a robust correlation with solver run time, leading to the development of a classifier capable of accurately distinguishing between easily solvable industrial instances and challenging random/crafted scenarios. We also present scaling studies of formulas with HCS structures to further support of theoretical analysis.

In the latter part of the thesis, the focus shifts to satisfaction-driven clause-learning (SDCL) solvers, known to be being exponentially more powerful than CDCL solvers. Despite the theoretical strength of SDCL, it remains a challenge to automate and determinize such solvers. To address this, we again leverage machine learning techniques to strategically

decide when to invoke an SDCL subroutine, with the goal of minimizing the associated overhead. The resulting SDCL solver, enhanced with MaxSAT techniques and conflict analysis, outperforms existing solvers on combinatorial benchmarks, particularly demonstrating superior efficacy on Mutilated Chess Board (MCB) problems.

Acknowledgments

Embarking on this doctoral journey has been a profound experience, and it would have been insurmountable without the support and guidance I received from various individuals.

First and foremost, I extend my deepest gratitude to my advisor, Vijay Ganesh, for his unwavering and patient guidance, invaluable feedback, and the countless hours spent discussing, and refining my research and making me a better learner, thinker and person. Your belief in my potential and your dedication to my academic growth have been pivotal in reaching this milestone.

I am indebted to the members of my thesis committee, for their insightful comments, constructive criticisms, and the challenging questions which motivate me to widen my research from various perspectives.

My heartfelt appreciation goes out to my colleagues and friends at the University of Waterloo, who provided an intellectually stimulating environment and were always available for discussions, debates, and much-needed breaks in between.

Completing this thesis has been a challenging yet rewarding journey, and each of you has been instrumental in guiding, supporting, and motivating me throughout. Thank you.

Dedication

This is dedicated to my family and those who have been by my side throughout my journey – words cannot express how grateful I am for all of the sacrifices you've made on my behalf. Your faith in my capabilities and endless love fueled my persistence in the face of challenges.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The primary focus of the research presented in this thesis is a deeper theoretical understanding of SAT solvers, and leveraging that understanding to empirically improving them further. SAT solvers have been widely used for a variety of applications, including problems in software engineering [29], formal methods [32], security [40, 117], and AI [24]. The problem solved by SAT solvers, namely, the Boolean satisfiability problem is at the core of the famous P vs. NP question [35]. Developing theories and tools for better understanding SAT algorithms is one of the most impactful lines of research within the community of researchers who are interested in the theory of solvers (broadly construed to include SAT, SMT, CP,VG MILP, and first-order automated theorem provers).

In this thesis, we developed deep insights into how SAT solvers work and what kinds of problems SAT solvers are good or bad at solving. The work and analysis are both theoretical and practical. On the theoretical side, we aim to understand what makes a solver robust and efficient for specific sets of problem benchmarks through the lens of proof complexity theory. First, this requires mathematical abstractions of solvers, that we use to perform rigorous theoretical analysis while at the same time empirically checking our theoretical understanding through performing extensive experiments. On the practical side, we aim at finding desirable sets of benchmarks for which SAT solvers are a natural fit for the problems, and then apply machine learning techniques to help solvers solve these classes of instances more efficiently. These ML techniques are usually innovated by a good understanding of proof theoretical properties of solvers and the class of instances under consideration.

There are four core chapters in this thesis. In Chapter 3, we prove a series of theoretical results that characterize the power of restarts for various models of SAT solvers [69]. More

precisely, we make the following contributions. First, we prove an exponential separation [1] between a *drunk* randomized CDCL solver model with restarts and the same model without restarts using a family of satisfiable instances. Second, we show that the configuration of CDCL solver with VSIDS branching and restarts (with activities erased after restarts) is exponentially more powerful than the same configuration without restarts for a family of unsatisfiable instances. To the best of our knowledge, these are the first separation results involving restarts in the context of SAT solvers. Third, we show that restarts do not add any proof complexity-theoretic power vis-a-vis several models of CDCL and DPLL solvers with non-deterministic static variable and value selection. Inspired by our theoretical model, we developed a practical restart policy which were able to solve all 500 instances of Bitcoin mining instances while many state-of-the-art solvers cannot solve any of the 500 instances at all.

In Chapter 4, taking the intuition developed in our theoretical analysis of the power of restarts, we propose a variant of restart, which we call reset. In most modern solvers, variable activities are preserved across restart boundaries, which means restarts do not interfere with the behavior of the branching heuristic. Hence solvers still branch "locally" with respect to the branching heuristic. To encourage more "global" reasoning, we study the effect of resets, which randomizes the activity scores of the variables of the input formula, thus potentially enabling a better *global exploration* of the search space. The crucial question one needs to address is when to invoke restarts with reset and when to invoke restarts without reset. We model the problem of whether to trigger restart with reset or restart without reset as a multi-armed bandit(MAB) problem and propose a reinforcement learning (RL) based reset policy using Thompson sampling. The Thompson sampling algorithm is designed to balance the exploration-exploitation tradeoff by probabilistically and adaptively choosing arms (restart with reset vs. restart with no reset) based on their estimated rewards. The proposed reset policy is compared with state-of-the-art CDCL solvers on a set of crypto benchmarks generated from bitcoin mining problems, and the results show that the proposed RL based reset policy outperforms baseline solvers MapleSAT, Kissat, Kissat_MAB_Hywalk. More specifically, all of these solvers can only solve less than 15 out of 500 of these crypto instances with a 5000 seconds timeout, whereas their modified version with our RL reset policy can solve almost all 500 instances given the same timeout. Additionally, our solver with RL reset remains competitive against their baseline on SAT competition instances.

In Chapter 5, for a better understanding of the success of modern CDCL solvers, we

[1] We say proof system A and proof system B are exponentially separated, or more specifically, A is exponentially more powerful than B, if there exists a class of formulas F, where F has polynomial size proofs in A and only exponential size proofs in B. See Background 2 for more details about these definitions.

proposed parameters based on a graph partitioning called Hierarchical Community Structure (HCS), which captures the recursive community structure of a graph of a Boolean formula [68]. The idea of HCS comes from the fact that human-written systems tend to be hierarchical and modular, and we expect Boolean formulas describing these systems will also have these structures. We show that HCS parameters are strongly correlative with solver run time using an Empirical Hardness Model, and further build a classifier based on HCS parameters that distinguishes between easy industrial and hard random/crafted instances with very high accuracy. We further strengthen our hypotheses via scaling studies. On the theoretical side, we show that counterexamples that plagued flat community structure do not apply to HCS and that there is a subset of HCS parameters such that restricting them limits the size of embeddable expanders.

In Chapter 6, we study a class of SAT solvers that are mathematically proven to be exponentially stronger than the state-of-the-art class of solvers for solving industrial SAT problems, namely conflict-driven clause learning (CDCL) solvers [80, 83]. CDCL based solvers have been dominating the field for more than 20 years. Despite its empirical success, it has been mathematically proven that there exists an infinite family of SAT problem inputs that would require any CDCL solvers exponential time (in the input formula's size) to solve, which means CDCL solvers are hopeless at solving such instances. Most of the work currently in the community aims at understanding and improving CDCL based solvers, however, it is also important to attempt to break the barrier of CDCL and study stronger systems. Satisfaction-driven clause-learning (SDCL) solvers are variants of CDCL solvers recently proposed, it was mathematically shown to be strictly more powerful than CDCL solvers [60, 59]. That is, there exist input formulas that require exponential CDCL runtime and only polynomial SDCL runtime. The idea of SDCL is still at its young stage and there has not been much work around it. We think the idea of SDCL is very natural, theoretically powerful, and interesting.

Even though SDCL solvers are theoretically more powerful than CDCL solvers, determinizing and designing a practical SDCL solver is still very challenging. It has been mathematically proven that determinizing and automating CDCL is NP-Hard and thus infeasible [9]. And we believe automating SDCL is only going to be harder as it is more powerful. But this also makes SDCL a very fruitful direction to explore, the idea we have is to use ML techniques to help automate SDCL solvers. At the end of the day, solvers are logical reasoning engines that apply logical deduction step by step, the bottleneck in automating solvers comes precisely from the fact that sequencing these deduction steps is hard. And we think reinforcement learning can be quite powerful in helping solvers select and sequence deduction steps. In our SDCL paper [90], where we designed a basic infrastructure of SDCL on top of MapleSAT. The infrastructure was combined with

a MaxSAT technique for learning SDCL clauses, as well as a technique inspired by conflict analysis which tries to minimize the final SDCL clause being learned by the solver. The final solver we designed outperforms all existing solvers over a class of combinatorial benchmarks called Multilated Chess Board (MCB) problems. To build on to this work, we design a ML technique for invoking SDCL to maximize the learning rate.

In summary, this thesis presents new proof complexity-theoretic analysis of SAT solver heuristics, specifically restart and reset, and leverages the insights thus gathered to develop reinforcement learning based reset and SDCL invocation policies. We also present extensive empirical evaluation of presented techniques on a large set of industrial benchmarks.

Chapter 2

Background

Below we provide relevant definitions and concepts used in this thesis. We refer the reader to the Handbook of Satisfiability [20] for literature on CDCL and DPLL solvers and to [64, 14] for literature on proof complexity.

2.1 CNF formulas

Let $\mathcal{X}$ be a finite set of propositional *variables*. A *literal* is a propositional variable (x) or the negation of one ($\neg x$). The *negation* of a literal l, denoted $\neg l$, is x if $l = \neg x$ and is $\neg x$ if $l = x$. A *clause* is a disjunction of distinct literals $l_1 \vee \ldots \vee l_n$ (interchangeably denoted with or without brackets). A *CNF formula* is a conjunction of distinct clauses $C_1 \wedge \ldots \wedge C_m$. When convenient, we consider a clause to be the set of its literals, and a CNF to be the set of its clauses. In the rest of the chapter, we assume that all formulas are CNF.

Satisfaction An *assignment* is a set of non-contradictory literals. A *total* assignment contains, for each variable $x \in \mathcal{X}$, either x or $\neg x$. Otherwise, it is a *partial* assignment. We denote by $\neg\alpha$ the clause consisting of the negation of all literals in the assignment α. An assignment α satisfies a literal l if $l \in \alpha$, it satisfies a clause C if it satisfies at least one of the literals in C, and it satisfies a formula F if it satisfies all the clauses in F. We denote these as $\alpha \models l$, $\alpha \models C$, and $\alpha \models F$, respectively. A *model* for a formula is an assignment that satisfies it. A formula with at least one model is *satisfiable*; otherwise, it is *unsatisfiable*. Given a formula F, the *SAT* problem consists of determining whether F is satisfiable. An assignment α falsifies a literal l if $\neg l \in \alpha$, falsifies a clause if it falsifies

all its literals, and falsifies a formula if it falsifies at least one of its clauses. The truth values of literals, clauses, and formulas are *undefined* for an assignment if they are neither falsified nor satisfied. Given a clause C and an assignment α, we denote by $touched_\alpha(C)$ the disjunction of all literals of C that are either satisfied or falsified by α, by $untouched_\alpha(C)$ the disjunction of all undefined literals, and by $satisfied_\alpha(C)$ the disjunction of all satisfied literals.

Unit propagation Given a formula F and an assignment α, unit propagation extends α by repeatedly applying the following rule until reaching a fixed point: if there is a clause with all literals falsified by α except one literal l, which is undefined, add l to α. If, as a result, a clause is found that is falsified by α (called *conflict*), the procedure stops and reports that a conflict clause has been found.

Formula relations Two formulas F and G are *equisatisfiable*, denoted $F \equiv_{SAT} G$, if F is satisfiable if and only if G is satisfiable, and they are *equivalent*, denoted $F \equiv G$, if they are satisfied by the same total assignments. We write $F \vdash_1 G$ (F *implies* G *by unit propagation*) if for every clause $C \in G$ of the form $l_1 \vee \ldots \vee l_n$, it holds that unit propagation applied to $F \wedge \neg l_1 \wedge \ldots \wedge \neg l_n$ results in a conflict. We say that G is a logical consequence of F (written $F \models G$) if all models of F are models of G.

CDCL The Conflict-Driven Clause Learning (CDCL) algorithm is the most successful procedure to-date for determining whether certain types of industrial formulas are satisfiable [79]. Let F denote such a formula. The CDCL procedure starts with an empty assignment α, which is extended and reduced in a last-in first-out (LIFO) way, by the following three steps until the satisfiability of the formula is determined (see Algorithm 6.2.2 removing lines 9-12):

1. Unit propagation is applied.

2. If a conflict is found, a *conflict analysis* procedure [119] derives a clause C (called a *lemma*) which is a logical consequence of F. If C is the empty clause, we can conclude that F is unsatisfiable. Otherwise, it is guaranteed that by removing enough literals from α, a new unit propagation is possible due to C. This process is called *backjump*. Additionally, lemma C is conjuncted (*learnt*) with F, and the procedure returns to step (i).

3. If no conflict is found in unit propagation, either α is a total assignment (and hence it satisfies the formula), or an undefined literal is chosen and added to α (the branching

step). The choice of this literal, called a *decision literal*, is determined by sophisticated heuristics [18] that can have a huge impact on the performance of the CDCL procedure.

CDCL as a Proof System We treat CDCL solvers as proof systems. For proof systems A and B, we use $A \sim_p B$ to denote that they are polynomially equivalent (p-equivalent). Throughout this chapter, it is convenient to think of the trail π of the solver during its run on a formula F as a restriction to that formula. We call a function $\pi\colon \{x_1, \ldots, x_n\} \to \{0, 1, *\}$ a *restriction*, where $*$ denotes that the variable is unassigned by π. A

2.2 Restart Policies

A restart policy is a method that erases part of the state of the solver at certain intervals during the run of a solver [48]. In most modern CDCL solvers, the restart policy erases the assignment trail upon invocation but may choose not to erase the learnt clause database or variable activities. Throughout this chapter, we assume that all restart policies are non-deterministic, i.e., the solver may (dynamically) non-deterministically choose its restart sequence. We refer the reader to a paper by Liang et al. [75] for a detailed discussion on modern restart policies.

To overcome the limitation of biased "local" search introduced by restarts, the concept of resetting has been proposed, where the activity scores of the variables are randomized after the assignment trail is erased [1]. Traditionally, a reset refers to zeroing out or randomizing activity scores of all variables of an input formula. Below we provide formal definitions for reset strategies, categorized as full and partial reset.

Definition 1 (Full Reset)**.** *Upon invocation, a full reset policy randomizes the activity scores of all variables (and thus randomizes the variable order for branching), in addition to deleting the contents of the assignment trail as in restart policies.*

Definition 2 ((k-)Partial Reset)**.** *Upon invocation, a partial reset policy retains the top k variables in the same exact order as before the reset, and the order of all the remaining variable activities is randomized. The contents of the assignment trail are deleted.*

[1] We are aware of empirical work on these, but we couldn't find formal papers on this topic

2.3 Reinforcement Learning

Reinforcement Learning (RL) is an area of machine learning that focuses on training agents to take actions (make decisions) in an environment to maximize some notion of reward. The agent learns through a process of interacting with the environment and receiving feedback in the form of rewards or penalties. RL frameworks are typically based on the idea of a Markov Decision Process (MDP), which is a mathematical model for decision making in which the outcomes of actions are probabilistic and depend on the current state of the environment. The agent's objective is to learn a policy, which is a mapping from states to actions, that maximizes the expected cumulative reward over time.

2.3.1 Multi-Armed Bandit (MAB)

The Multi-Armed Bandit (MAB) problem is one of the simplest RL problems and is particularly useful when we want to model an agent that has to make frequent actions in a dynamic and uncertain environment. In MAB problems, an agent takes action and receives feedback from the environment. Over time, the agent learns which action (or combination of actions) is likely to provide the best reward in the long run and selects it more frequently.

MAB problems exhibit a nice trade-off between exploration and exploitation as the agent has to constantly make choices to decide which arm to play. Selecting a less explored arm to play is considered exploration, while repeatedly playing a previously played arm is considered exploitation. After a sufficient amount of exploration, an RL agent may have enough confidence in the expected rewards from different arms that it can start exploiting the arms with the highest expected reward. We refer the reader to Sutton and Barto for a detailed discussion on reinforcement learning and MAB problems [108].

2.3.2 Thompson Sampling

Thompson sampling [110], also called Probability matching strategies or Bayesian Bandits is a well-known approach for solving the MAB problem. In Thompson Sampling, the algorithm assumes that the prior distribution of each arm's(action's) reward is a beta distribution. The beta distribution has two shape parameters, denoted by α and β, which control the shape of the distribution. The mean of the distribution is given by $\alpha/(\alpha+\beta)$, and the variance is given by $\alpha\beta/((\alpha+\beta)^2(\alpha+\beta+1))$. In the context of Thompson Sampling, the shape parameters α and β are often used to represent the number of successes and failures of each arm, respectively. Typically, we would consider pulling an arm as a success

if the reward from pulling the arm is "good enough", what is "good enough" depends on the context and should be customized to adapt to the underlying problem.

Each time an arm is pulled, the algorithm updates the shape parameters for the beta distribution of that arm based on the outcome of the pull. If the pull results in a success, α for that arm is incremented by 1, and if it fails, the arm's β is incremented by 1. These updates help to refine the algorithm's estimate of the reward distribution for each arm, which in turn informs its future decisions about which arm to pull. By using the beta distribution in this way, Thompson Sampling can balance the exploration-exploitation tradeoff by probabilistically selecting arms based on their estimated reward distributions. This approach allows the algorithm to explore arms with uncertain rewards while also exploiting arms with high estimated rewards, leading to better performance.

2.4 Variable Incidence Graph (VIG)

Researchers have proposed a variety of graphs to study graph-theoretic properties of Boolean formulas. In this work, we focus on the Variable Incidence Graph (VIG), primarily due to the relative ease of computing community structure over VIGs compared to other graph representations. The VIG for a formula F over variables $x_1, \ldots, x_n$ has n vertices, one for each variable. There is an edge between vertices x_i and x_j if both x_i and x_j occur in some clause C_k in F. One drawback of VIGs is that a clause of width w corresponds to a clique of size w in the VIG. Therefore, large width clauses (of size n^{ε}) can significantly distort the structure of a VIG, and formulas with such large width clauses should have their width reduced (via standard techniques) before using a VIG.

2.5 Community Structure and Modularity

Intuitively, a set of variables (vertices in the VIG) of a formula forms a community if these variables are more densely connected than to variables outside of the set. An (optimal) community structure of a graph is a partition $P = \{V_1, \ldots, V_k\}$ of its vertices into communities that optimizes some measure capturing this intuition, for instance, modularity [88], which is the one we use in this chapter. Let $G = (V, E)$ be a graph with adjacency matrix A and for each vertex $v \in V$ denote by $d(v)$ its degree. Let $\delta_P \colon V \times V \to \{0, 1\}$ be the community indicator function of a partition, i.e. $\delta_P(u, v) = 1$ iff vertices u and v belong to

the same community in P. The *modularity* of the partition P is

$$Q(P) := \frac{1}{2|E|} \sum_{u,v \in V} \left[A_{u,v} - \frac{d(u)d(v)}{2|E|} \right] \delta_P(u,v) \tag{2.1}$$

Note that $Q(P)$ ranges from -0.5 to 1, with values close to 1 indicating good community structure. We define the modularity $Q(G)$ of graph G as the maximum modularity over all possible partitions, with corresponding partition $\mathcal{P}(G)$. Other measures may produce radically different partitions.

2.6 Expansion of a Graph

Expansion is a measure of graph connectivity [61]. Out of several equivalent such measures, the most convenient to relate to HCS is *edge expansion*: given a subset of vertices $S \subseteq V$, its edge expansion is $h(S) = |E(S, V \backslash S)| \,/\, |S|$, and the edge expansion of a graph is $h(G) = \min_{1 \leq |S| \leq n/2} h(S)$. A graph family G_n is an expander if $h(G_n)$ is bounded away from zero. Resolution lower bounds (of both random and crafted formulas) often rely on strong expansion properties of the graph [15].

Chapter 3

Towards a Complexity-theoretic Understanding of Restarts in SAT solvers

Restarts are a widely-used class of techniques integral to the efficiency of Conflict-Driven Clause Learning (CDCL) Boolean SAT solvers. While the utility of such policies has been well-established empirically, a theoretical explanation of whether restarts are indeed crucial to the power of CDCL solvers is lacking. In this chapter, we prove a series of theoretical results that characterize the power of restarts for various models of SAT solvers. More precisely, we make the following contributions. First, we prove an exponential separation between a *drunk* randomized CDCL solver model with restarts and the same model without restarts using a family of satisfiable instances. Second, we show that the configuration of CDCL solver with VSIDS branching and restarts (with activities erased after restarts) is exponentially more powerful than the same configuration without restarts for a family of unsatisfiable instances. To the best of our knowledge, these are the first separation results involving restarts in the context of SAT solvers. Third, we show that restarts do not add any proof complexity-theoretic power vis-a-vis several models of CDCL and DPLL solvers with non-deterministic static variable and value selection.

3.1 Introduction

Over the last two decades, Conflict-Driven Clause Learning (CDCL) SAT solvers have had a revolutionary impact on many areas of software engineering, security, and AI. This is

primarily due to their ability to solve real-world instances containing millions of variables and clauses [80, 83, 20, 93, 7], even though the Boolean SAT problem is known to be an NP-complete problem and is believed to be intractable in the worst case.

This remarkable success has prompted complexity theorists to seek an explanation for the efficacy of CDCL solvers, to bridge the gap between theory and practice. Fortunately, a few results have already been established that lay the groundwork for a deeper understanding of SAT solvers viewed as proof systems [13, 53, 27]. Among them, the most important result is the one by Pipatsrisawat and Darwiche [93] and independently by Atserias et al. [7], which shows that an idealized model of CDCL solvers with non-deterministic branching (variable selection and value selection), and restarts is *polynomially equivalent* to the general resolution proof system. However, an important question that remains open is whether this result holds even when restarts are disabled, i.e., whether configurations of CDCL solvers without restarts (when modeled as proof systems) are polynomial equivalent to the general resolution proof system. In practice, there is significant evidence that restarts are crucial to solver performance.

This question of the "power of restarts" has prompted considerable theoretical work. For example, Bonet, Buss, and Johannsen [25] showed that CDCL solvers with no restarts (but with non-deterministic variable and value selection) are strictly more powerful than regular resolution. Despite this progress, the central questions, such as whether restarts are integral to the efficient simulation of general resolution by CDCL solvers, remain open.

In addition to the aforementioned theoretical work, there have been many empirical attempts at understanding restarts given how important they are to solver performance. Many hypotheses have been proposed aimed at explaining the power of restarts. Examples include the *heavy-tail* explanation [48], and the "restarts compact assignment trail and hence produce clauses with lower literal block distance (LBD)" perspective [75]. Having said that, the heavy-tailed distribution explanation of the power of restarts is not considered valid anymore in the CDCL setting [75].

3.1.1 Contributions

In this chapter, we make several contributions to the theoretical understanding of the power of restarts for several restricted models of CDCL solvers:

1. First, we show that CDCL solvers with backtracking, non-deterministic dynamic

variable selection, randomized value selection, and restarts[1] are exponentially faster than the same model, but without restarts, with high probability (w.h.p)[2]. A notable feature of our proof is that we obtain this separation on a family of satisfiable instances. (See Section 3.3 for details.)

2. Second, we prove that CDCL solvers with VSIDS variable selection, phase saving value selection, and restarts (where activities of variables are reset to zero after restarts) are exponentially faster (w.h.p) than the same solver configuration but without restarts for a class of unsatisfiable formulas. This result holds irrespective of whether the solver uses backtracking or backjumping. (See Section 3.4 for details.)

3. Finally, we prove several smaller separation and equivalence results for various configurations of CDCL and DPLL solvers with and without restarts. For example, we show that CDCL solvers with non-deterministic static variable selection, non-deterministic static value selection, and with restarts, are polynomially equivalent to the same model but without restarts. Another result we show is that for DPLL solvers, restarts do not add proof theoretic power as long as the solver configuration has non-deterministic dynamic variable selection. (See Section 3.5 for details.)

3.2 Notation for Solver Configurations Considered

In this section, we precisely define the various heuristics used to define SAT solver configurations in this paper. By the term *solver configuration*, we mean a solver parameterized with appropriate heuristic choices. For example, a CDCL solver with non-deterministic variable and value selection, as well as an asserting learning scheme with restarts would be considered a solver configuration.

To keep track of these configurations, we denote solver configurations by the notation $M^{E,R}_{A,B}$, where M indicates the underlying *solver model* (we use C for CDCL and D for DPLL solvers); the subscript A denotes a *variable selection scheme*; the subscript B is a *value selection scheme*; the superscript E is a *backtracking scheme*, and finally the superscript R indicates whether the solver configuration comes equipped with a restart policy. That is, the presence of the superscript R indicates that the configuration has restarts, and its absence indicates that it does not. A $*$ in place of A, B, or E denotes that the scheme is

[1] In keeping with the terminology from [1], we refer to any CDCL solver with randomized value selection as a *drunk* solver.

[2] We say that an event occurs with high probability (w.h.p.) if the probability of that event happening goes to 1 as $n \to \infty$.

Table 3.1: Solver configurations in the order they appear. ND stands for non-deterministic dynamic.

	Model	Variable Selection	Value Selection	Backtracking	Restarts
$C^{T,R}_{ND,RD}$	CDCL	ND	Random Dynamic	Backtracking	Yes
$C^{T}_{ND,RD}$	CDCL	ND	Random Dynamic	Backtracking	No
$C^{J,R}_{VS,PS}$	CDCL	VSIDS	Phase Saving	Backjumping	Yes
$C^{J}_{VS,PS}$	CDCL	VSIDS	Phase Saving	Backjumping	No
$C^{J,R}_{S,S}$	CDCL	Static	Static	Backjumping	Yes
$C^{J}_{S,S}$	CDCL	Static	Static	Backjumping	No
$D^{T}_{ND,*}$	DPLL	ND	Arbitrary	Backtracking	No
$D^{T,R}_{ND,ND}$	DPLL	ND	ND	Backtracking	Yes
$D^{T}_{ND,ND}$	DPLL	ND	ND	Backtracking	No
$D^{T,R}_{ND,RD}$	DPLL	ND	Random Dynamic	Backtracking	Yes
$D^{T}_{ND,RD}$	DPLL	ND	Random Dynamic	Backtracking	No
$C^{J,R}_{ND,ND}$	CDCL	ND	ND	Backjumping	Yes
$C^{J}_{ND,ND}$	CDCL	ND	ND	Backjumping	No

arbitrary, meaning that it works for any such scheme. See Table 3.1 for examples of solver configurations studied in this chapter.

3.2.1 Variable Selection Schemes

1. Static (S): Upon invocation, the S variable selection heuristic returns the unassigned variable with the highest rank according to some predetermined, fixed, total ordering of the variables.

2. Non-deterministic Dynamic (ND): The ND variable selection scheme non-deterministically selects and returns an unassigned variable.

3. VSIDS (VS) [83]: Each variable has an associated number, called its *activity*, initially set to 0. Each time a solver learns a conflict, the activities of variables appearing on the conflict side of the implication graph receive a constant bump. The activities of all variables are decayed by a constant c, where $0 < c < 1$, at regular intervals. The VSIDS variable

selection heuristic returns the unassigned variable with the highest activity, with ties broken randomly.

3.2.2 Value Selection Schemes

1. Static (S): Before execution, a 1-1 mapping of variables to values is fixed. The S value selection heuristic takes as input a variable and returns the value assigned to that variable according to the predetermined mapping.

2. Non-deterministic Dynamic (ND): The ND value selection scheme non-deterministically selects and returns a truth assignment.

3. Random Dynamic (RD): A randomized algorithm that takes as input a variable and returns a uniformly random truth assignment.

4. Phase Saving (PS): A heuristic that takes as input an unassigned variable and returns the previous truth value that was assigned to the variable. Typically solver designers determine what value is returned when a variable has not been previously assigned. For simplicity, we use the phase saving heuristic that returns 0 if the variable has not been previously assigned.

3.2.3 Backtracking and Backjumping Schemes

To define different backtracking schemes we use the concept of *decision level* of a variable x, which is the number of decision variables on the trail prior to x. **Backtracking (T):** Upon deriving a conflict clause, the solver undoes the most recent decision variable on the assignment trail. **Backjumping (J):** Upon deriving a conflict clause, the solver undoes all decision variables with a decision level higher than the variable with the second-highest decision level in the conflict clause.

Note on Solver Heuristics. Most of our results hold irrespective of the choice of deterministic asserting clause learning schemes (except for Proposition 6). Additionally, the questions we address in this chapter make sense only when it is assumed that solver heuristics are polynomial time methods.

3.3 Separation for Drunk CDCL with and without Restarts

Inspired by Alekhnovich et al. [1], where the authors proved an exponential lower bound for drunk DPLL solvers over a class of satisfiable instances, we studied the behavior of restarts in a drunken model of CDCL solver. We introduce a class of satisfiable formulas, $Ladder_n$, and use them to prove the separation between $C^{T,R}_{ND,RD}$ and $C^{T}_{ND,RD}$. At the core of these formulas is a formula that is hard for general resolution even after any small restriction (corresponding to the current trail of the solver). For this, we use the well-known Tseitin formulas.

Definition 3 (Tseitin Formulas)**.** *Let $G = (V, E)$ be a graph and $f\colon V \to \{0, 1\}$ a labelling of the vertices. The formula $Tseitin(G, f)$ has variables x_e for $e \in E$ and constraints $\bigoplus_{uv\in E} x_{uv} = f(v)$ for each $v \in V$.*

For any graph G, $Tseitin(G, f)$ is unsatisfiable iff $\bigoplus_{v\in V} f(v) = 1$, in which case we call f an *odd labelling*. The specifics of the labelling are irrelevant for our applications, any odd labelling will do. Therefore, we often omit defining f, and simply assume that it is odd.

The family of satisfiable $Ladder_n$ formulas are built around the Tseitin formulas, unless the variables of the formula are set to be consistent to one of two satisfying assignments, the formula will become unsatisfiable. Furthermore, the solver will only be able to backtrack out of the unsatisfiable sub-formula by first refuting Tseitin, which is a provably hard task for any CDCL solver [112].

The $Ladder_n$ formulas contain two sets of variables, ℓ^i_j for $0 \le i \le n-2, j \in [\log n]$ and c_m for $m \in [\log n]$, where n is a power of two. We denote by ℓ^i the block of variables $\{\ell^i_1, \ldots, \ell^i_{\log n}\}$. These formulas are constructed using the following gadgets.

Ladder gadgets: $L^i := (\ell^i_1 \vee \ldots \vee \ell^i_{\log n}) \wedge (\neg\ell^i_1 \vee \ldots \vee \neg\ell^i_{\log n})$.
Observe that L^i is falsified only by the all-1 and all-0 assignments.

Connecting gadgets: $C^i := (c_1^{bin(i,1)} \wedge \ldots \wedge c_{\log n}^{bin(i,\log n)})$.
Here, $bin(i, m)$ returns the mth bit of the binary representation of i, and $c^1_m := c_m$, while $c^0_m := \neg c_m$. That is, C^i is the conjunction that is satisfied only by the assignment encoding i in binary.

Equivalence gadget: $EQ := \bigwedge_{i,j=0}^{n-2} \bigwedge_{m,k=1}^{\log n} (\ell^i_k \iff \ell^j_m)$.
These clauses enforce that every ℓ-variable must take the same value.

Definition 4 (Ladder formulas)**.** *For $G = (V, E)$ with $|E| = n - 1$ where n is a power of two, let $Tseitin(G, f)$ be defined on the variables $\{\ell_1^0, \ldots, \ell_1^{n-2}\}$. $Ladder_n(G, f)$ is the conjunction of the clauses representing*

$$\begin{aligned} &L^i \Rightarrow C^i, && \forall 0 \leq i \leq n-2 \\ &C^i \Rightarrow Tseitin(G, f), && \forall 0 \leq i \leq n-2 \\ &C^{n-1} \Rightarrow EQ. \end{aligned}$$

Observe that the $Ladder_n(G, f)$ formulas have polynomial size provided that the degree of G is $O(\log n)$. As well, this formula is satisfiable only by the assignments that sets $c_m = 1$ and $\ell_j^i = \ell_q^p$ for every $m, j, q \in [\log n]$ and $0 \leq i, p \leq n - 2$.

These formulas are constructed so that after setting only a few variables, any drunk solver will enter an unsatisfiable subformula w.h.p. and thus be forced to refute the Tseitin formula. Both the ladder gadgets and equivalence gadget act as trapdoors for the Tseitin formula. Indeed, if any c-variable is set to 0 then we have already entered an unsatisfiable instance. Similarly, setting $\ell_j^i = 1$ and $\ell_q^p = 0$ for any $0 \leq i, p \leq n - 2$, $j, q \in [\log n]$ causes us to enter an unsatisfiable instance. This is because setting all c-variables to 1 together with this assignment would falsify a clause of the equivalence gadget. Thus, after the second decision of the solver, the probability that it is in an unsatisfiable instance is already at least 1/2. With these formulas in hand, we prove the following theorem, separating backtracking $C^T_{ND,RD}$ solvers with and without restarts.

Theorem 1. *There exists a family of $O(\log n)$-degree graphs G such that*

1. *$Ladder_n(G, f)$ can be decided in time $O(n^2)$ by $C^{T,R}_{ND,RD}$, except with exponentially small probability.*
2. *$C^T_{ND,RD}$ requires exponential time to decide $Ladder_n(G, f)$, except with probability $O(1/n)$.*

The proof of the preceding theorem occupies the remainder of this section.

3.3.1 Upper Bound on Ladder Formulas Via Restarts.

We present the proof for part (1) of Theorem 1. The proof relies on the following lemma, stating that given the all-1 restriction to the c-variables, $C^T_{ND,RD}$ will find a satisfying assignment.

Lemma 1. *For any graph G, $C^{T}_{ND,RD}$ will find a satisfying assignment to $Ladder_n(G, f)[c_1 = 1, \ldots, c_{\log n} = 1]$ in time $O(n \log n)$.*

Proof. When all c variables are 1, we have $C^{n-1} = 1$. By the construction of the connecting gadget, $C^i = 0$ for all $0 \leq i \leq n-2$. Under this assignment, the remaining clauses belong to EQ, along with $\neg L^i$ for $0 \leq i \leq n-2$. It is easy to see that, as soon as the solver sets an ℓ-variable, these clauses will propagate the remaining ℓ-variables to the same value. □ □

Put differently, the set of c variables forms a *weak backdoor* [115, 116] for $Ladder_n$ formulas. Part (1) of Theorem 1 shows that, with probability at least 1/2, $C^{T,R}_{ND,RD}$ can exploit this weak backdoor using only $O(n)$ number of restarts.

of Theorem 1 Part (1). By Lemma 1, if $C^{T,R}_{ND,RD}$ is able to assign all c variables to 1 before assigning any other variables, then the solver will find a satisfying assignment in time $O(n \log n)$ with probability 1. We show that the solver can exploit restarts in order to find this assignment. The strategy the solver adopts is as follows: query each of the c-variables; if at least one of the c-variables was assigned to 0, restart. We argue that if the solver repeats this procedure $k = n^2$ times then it will find the all-1 assignment to the c-variables, except with exponentially small probability. Because each variable is assigned 0 and 1 with equal probability, the probability that a single round of this procedure finds the all-1 assignment is $2^{-\log n}$. Therefore, the probability that the solver has not found the all-1 assignment after k rounds is

$$(1 - 1/n)^k \leq e^{-k/n} = e^{-n}.$$ □

□

3.3.2 Lower Bound on Ladder Formulas Without Restarts

We now prove part (2) of Theorem 1. The proof relies on the following three technical lemmas. The first claims that the solver is well-behaved (most importantly that it cannot learn any new clauses) while it has not made many decisions.

Lemma 2. *Let G be any graph of degree at least d. Suppose that $C^{T}_{ND,RD}$ has made $\delta < \min(d-1, \log n - 1)$ decisions since its invocation on $Ladder_n(G, f)$. Let π_δ be the current trail, then*

1. *The solver has yet to enter a conflict, and thus has not learned any clauses.*

2. The trail π_δ contains variables from at most δ different blocks ℓ^i.

The proof of this lemma is deferred to the appendix.

The following technical lemma states that if a solver with backtracking has caused the formula to become unsatisfiable, then it must *refute* that formula before it can backtrack out of it. For a restriction π and a formula F, we say that the solver has *produced a refutation* of an unsatisfiable formula $F[\pi]$ if it has learned a clause C such that C is falsified under π. Note that because general resolution p-simulates CDCL, any refutation of a formula $F[\pi]$ implies a general resolution refutation of $F[\pi]$ of size at most polynomial in the time that the solver took to produce that refutation.

Lemma 3. *Let F be any propositional formula, let π be the current trail of the solver, and let x be any literal in π. Then, $C^T_{ND,ND}$ backtracks x only after it has produced a refutation of $F[\pi]$.*

Proof. In order to backtrack x, the solver must have learned a clause C asserting the negation of some literal $z \in \pi$ that was set before x. Therefore, C must only contain the negation of literals in π. Hence, $C[\pi] = \emptyset$. □ □

The third lemma reduces proving a lower bound on the runtime of $C^T_{ND,ND}$ on the $Ladder_n$ formulas under any well-behaved restriction to proving a general resolution lower bound on an associated Tseitin formula.

Definition 5. *For any unsatisfiable formula F, denote by $Res(F \vdash \emptyset)$ the minimal size of any general resolution refutation of F.*

We say that a restriction (thought of as the current trail of the solver) π to $Ladder_n(G, f)$ *implies Tseitin* if π either sets some c-variable to 0 or $\pi[\ell^i_j] = 1$ and $\pi[\ell^p_q] = 0$ for some $0 \leq i, q \leq n-2$, $j, q \in [\log n]$. Observe that in both of these cases the formula $Ladder_n(G, f)[\pi]$ is unsatisfiable.

Lemma 4. *Let π be any restriction that implies Tseitin and such that each clause of $Ladder_n(G, f)[\pi]$ is either satisfied or contains at least two unassigned variables. Suppose that π sets variables from at most δ blocks ℓ^i. Then there is a restriction ρ^*_π that sets at most δ variables of $Tseitin(G, f)$ such that*

$$Res(Ladder_n(G, f)[\pi] \vdash \emptyset) \geq Res(Tseitin(G, f)[\rho^*_\pi] \vdash \emptyset).$$

We defer the proof of this lemma to the appendix, and show how to use them to prove part (2) of Theorem 1. We prove this statement for any degree $O(\log n)$ graph G with sufficient expansion.

Definition 6. *The expansion of a graph $G = (V, E)$ is*

$$e(G) := \min_{V' \subseteq V, |V'| \leq |V|/2} \frac{|E[V', V \setminus V']|}{|V'|},$$

where $E[V', V \setminus V']$ is the set of edges in E with one endpoint in V' and the other in $V \setminus V'$.

For every $d \geq 3$, *Ramanujan Graphs* provide an infinite family of d-regular expander graphs G for which $e(G) \geq d/4$. The lower bound on solver runtime relies on the general resolution lower bounds for the Tseitin formulas [112]; we use the following lower bound criterion which follows immediately[3] from [15].

Corollary 1 ([15])**.** *For any connected graph $G = (V, E)$ with maximum degree d and odd weight function f,*

$$Res(Tseitin(G, f) \vdash \emptyset) = \exp\left(\Omega\left(\frac{(e(G)|V|/3 - d)^2}{|E|}\right)\right)$$

We are now ready to prove the theorem.

of part (2) Theorem 1. Fix $G = (V, E)$ to be any degree-$(8 \log n)$ graph on $|E| = n - 1$ edges such that $e(G) \geq 2 \log n$. Ramanujan graphs satisfy these conditions.

First, we argue that within $\delta < \log n - 1$ decisions from the solver's invocation, the trail π_δ will imply Tseitin, except with probability $1 - /2^{\delta - 1}$. By Lemma 2, the solver has yet to backtrack or learn any clauses, and it has set variables from at most δ blocks ℓ^i. Let x be the variable queried during the δth decision. If x is a c variable, then with probability $1/2$ the solver sets $c_i = 0$. If x is a variable ℓ^i_j, then, unless this is the first time the solver sets an ℓ-variable, the probability that it sets ℓ^i_j to a different value than the previously set ℓ-variable is $1/2$.

Conditioning on the event that, within the first $\log n - 2$ decisions the trail of the solver implies Tseitin (which occurs with probability at least $(n - 8)/n$), we argue that the

[3] In particular, this follows from Theorem 4.4 and Corollary 3.6 in [15], noting that the definition of expansion used in their paper is lower bounded by $3e(G)/|V|$ as they restrict to sets of vertices of size between $|V|/3$ and $2|V|/3$.

runtime of the solver is exponential in n. Let $\delta < \log n - 1$ be the first decision level such that the current trail π_δ implies Tseitin. By Lemma 3 the solver must have produced a refutation of $Ladder_n(G, f)[\pi_\delta]$ in order to backtrack out of the unsatisfying assignment. If the solver takes t steps to refute $Ladder_n(G, f)[\pi_\delta]$ then this implies a general resolution refutation of size $\mathsf{poly}(t)$. Therefore, in order to lower bound the runtime of the solver, it is enough to lower bound the size of general resolution refutations of $Ladder_n(G, f)[\pi_\delta]$.

By Lemma 2, the solver has not learned any clauses, and has yet to enter into a conflict and therefore no clause in $Ladder_n(G, f)[\pi_\delta]$ is falsified. As well, π_δ sets variables from at most $\delta < \log n - 1$ blocks ℓ^i. By Lemma 4 there exists a restriction ρ^*_π such that $Res(Ladder_n(G, f)[\pi] \vdash \emptyset) \geq Res(Tseitin(G, f)[\rho^*_\pi] \vdash \emptyset)$. Furthermore, ρ^*_π sets at most $\delta < \log n - 1$ variables and therefore cannot falsify any constraint of $Tseitin(G, f)$, as each clause depends on $8 \log n$ variables. Observe that if we set a variable x_e of $Tseitin(G, f)$ then we obtain a new instance of $Tseitin(G_{\rho^*_\pi}, f')$ on a graph $G_{\rho^*_\pi} = (V, E \setminus \{e\})$. Therefore, we are able to apply Corollary 1 provided that we can show that $e(G_{\rho^*_\pi})$ is large enough.

Claim 1. *Let $G = (V, E)$ be a graph and let $G' = (V, E')$ be obtained from G by removing at most $e(G)/2$ edges. Then $e(G') \geq e(G)/2$.*

Proof. Let $V' \subseteq V$ with $|V'| \leq |V|/2$. Then, $E'[V', V \setminus V'] \geq e(G)|V'| - e(G)/2 \geq (e(G)/2)|V'|$. □

It follows that $e(G_{\rho^*_\pi}) \geq \log n$. Note that $|V| = n/8 \log n$. By Corollary 1,

$$Res(Ladder_n(G, f)[\pi] \vdash \emptyset) = \exp(\Omega(((n-1)/24 - 8 \log n)^2/n)) = \exp(\Omega(n)).$$

Therefore, the runtime of $C^T_{ND,ND}$ is $\exp(\Omega(n))$ on $Ladder_n(G, F)$ w.h.p.

□

3.4 CDCL+VSIDS Solvers with and without Restarts

In this section, we prove that CDCL solvers with VSIDS variable selection, phase saving value selection and restarts (where activities of variables are reset to zero after restarts) are exponentially more powerful than the same solver configuration but without restarts, w.h.p.

Theorem 2. *There is a family of unsatisfiable formulas that can be decided in polynomial time with $C^{J,R}_{VS,PS}$ but requires exponential time with $C^J_{VS,PS}$, except with exponentially small probability.*

We show this separation using pitfall formulas $\Phi(G_n, f, n, k)$, designed to be hard for solvers using the VSIDS heuristic [114]. We assume that G_n is a constant-degree expander graph with n vertices and m edges, $f\colon V(G_n) \to \{0,1\}$ is a function with odd support as with Tseitin formulas, we think of k as a constant and let n grow. We denote the indicator function of a Boolean expression B with $[\![B]\!]$. These formulas have k blocks of variables named X_j, Y_j, Z_j, P_j, and A_j, with $j \in [k]$, and the following clauses:

- $\left(\bigoplus_{e \ni v} x_{j,e} = f(v)\right) \vee \bigvee_{i=1}^{n} z_{j,i}$, expanded into CNF, for $v \in V(G_n)$ and $j \in [k]$;
- $y_{j,i_1} \vee y_{j,i_2} \vee \neg p_{j,i_3}$ for $i_1, i_2 \in [n]$, $i_1 < i_2$, $i_3 \in [m+n]$, and $j \in [k]$;
- $y_{j,i_1} \vee \bigvee_{i \in [m+n] \setminus \{i_2\}} p_{j,i} \vee \bigvee_{i=1}^{i_2-1} x_{j,i} \vee \neg x_{j,i_2}$ for $i_1 \in [n]$, $i_2 \in [m]$, and $j \in [k]$;
- $y_{j,i_1} \vee \bigvee_{i \in [m+n] \setminus \{m+i_2\}} p_{j,i} \vee \bigvee_{i=1}^{m} x_{j,i} \vee \bigvee_{i=1+[\![i_2=n]\!]}^{i_2-1} z_{j,i} \vee \neg z_{j,i_2}$ for $i_1, i_2 \in [n]$ and $j \in [k]$;
- $\neg a_{j,1} \vee a_{j,3} \vee \neg z_{j,i_1}$, $\neg a_{j,2} \vee \neg a_{j,3} \vee \neg z_{j,i_1}$, $a_{j,1} \vee \neg z_{j,i_1} \vee \neg y_{j,i_2}$, and $a_{j,2} \vee \neg z_{j,i_1} \vee \neg y_{j,i_2}$ for $i_1, i_2 \in [n]$ and $j \in [k]$; and
- $\bigvee_{j \in [k]} \neg y_{j,i} \vee \neg y_{j,i+1}$ for odd $i \in [n]$.

To give a brief overview, the first type of clauses are essentially a Tseitin formula and thus are hard to solve. The next four types form a pitfall gadget, which has the following easy-to-check property.

Claim 2. *Given any pair of variables y_{j,i_1} and y_{j,i_2} from the same block Y_j, assigning $y_{j,i_1} = 0$ and $y_{j,i_2} = 0$ yields a conflict.*

Furthermore, such a conflict involves all of the variables of a block X_j, which makes the solver prioritize these variables and it becomes stuck in a part of the search space where it must refute the first kind of clauses. Proving this formally requires a delicate argument, but we can use the end result as a black box.

Theorem 3 ([114, Theorem 3.6])**.** *For k fixed, $\Phi(G_n, f, n, k)$ requires time $\exp(\Omega(n))$ to decide with $C^{J}_{VS,PS}$, except with exponentially small probability.*

The last type of clauses, denoted by Γ_i, ensure that a short general resolution proof exists. Not only that, we can also prove that pitfall formulas have small backdoors [115, 116], which is enough for a formula to be easy for $C^{J,R}_{VS,PS}$.

Definition 7. *A set of variables V is a strong backdoor for unit-propagation if every assignment to all variables in V leads to a conflict, after unit propagation.*

Lemma 5. *If F has a strong backdoor for unit-propagation of size c, then $C^{J,R}_{VS,PS}$ can solve F in time $n^{O(c)}$, except with exponentially small probability.*

Proof. We say that the solver learns a beneficial clause if it only contains variables in V. Since there are 2^c possible assignments to variables in V and each beneficial clause forbids at least one assignment, it follows that learning 2^c beneficial clauses is enough to produce a conflict at level 0.

Therefore it is enough to prove that, after each restart, we learn a beneficial clause with large enough probability. Since all variables are tied, all decisions before the first conflict after a restart are random, and hence with probability at least n^{-c} the first variables to be decided before reaching the first conflict are (a subset of) V. If this is the case then, since V is a strong backdoor, no more decisions are needed to reach a conflict, and furthermore all decisions in the trail are variables in V, hence the learned clause is beneficial.

It follows that the probability of having a sequence of n^{2c} restarts without learning a beneficial clause is at most

$$(1 - n^{-c})^{n^{2c}} \leq \exp(-n^{-c} \cdot n^{2c}) = \exp(-n^c) \tag{3.1}$$

hence by a union bound the probability of the algorithm needing more than $2^c \cdot n^{2c}$ restarts is at most $2^c \cdot \exp(-n^c)$. □

We prove Theorem 2 by showing that $\Phi(G_n, f, n, k)$ contains a backdoor of size $2k(k+1)$.

of Theorem 2. We claim that the set of variables $V = \{y_{j,i} \mid (j,i) \in [k] \times [2k+2]\}$ is a strong backdoor for unit-propagation. Consider any assignment to V. Each of the $k+1$ clauses $\Gamma_1, \Gamma_3, \ldots, \Gamma_{2k+1}$ forces a different variable $y_{j,i}$ to 0, hence by the pigeonhole principle there is at least one block with two variables assigned to 0. But by Claim 2, this is enough to reach a conflict.

The upper bound follows from Lemma 5, while the lower bound follows from Theorem 3. □

3.5 Minor Equivalences and Separations Solvers with and without Restarts

In this section, we prove four smaller separation and equivalence results for various configurations of CDCL and DPLL solvers with and without restarts.

3.5.1 Equivalence between CDCL Solvers with Static Configurations with and without Restarts

First, we show that CDCL solvers with non-deterministic static variable and value selection without restarts ($C^{J}_{S,S}$) is as powerful as the same configuration with restarts ($C^{J,R}_{S,S}$) for both satisfiable and unsatisfiable formulas. We assume that the BCP subroutine for the solver configurations under consideration is "fixed" in the following sense: if there is more than one unit clause under a partial assignment, the BCP subroutine propagates the clause that is added to the clause database first.

Theorem 4. *$C^{J}_{S,S} \sim_p C^{J,R}_{S,S}$ provided that they are given the same variable ordering and fixed mapping of variables to values for the variable selection and value selection schemes respectively.*

We prove this theorem by arguing for any run of $C^{J,R}_{S,S}$, that restarts can be removed without increasing the run-time.

Proof. Consider a run of $C^{J,R}_{S,S}$ on some formula F, and suppose that the solver has made t restart calls. Consider the trail π for $C^{J,R}_{S,S}$ up to the variable l from the second highest decision from the last learnt clause before the first restart call. Now, observe that because the decision and variable selection orders are static, once $C^{J,R}_{S,S}$ restarts, it will force it to repeat the same decisions and unit propagations that brought it to the trail π. Suppose that this is not the case and consider the first literal on which the trails differ. This difference could not be caused by a unit propagation as the solver has not learned any new clauses since the restart. Thus, it must have been caused by a decision. However, because the clause databases are the same, this would contradict the static variable and value order. Therefore, this restart can be ignored, and we obtain a run of $C^{J,R}_{S,S}$ with $d-1$ restarts without increasing the run-time. The proof follows by induction. Once all restarts have been removed, the result is a valid run of $C^{J}_{S,S}$. □

Note that in the proof of Theorem 4, not only we argue that $C^{J}_{S,S}$ is p-equivalent to $C^{J,R}_{S,S}$, we also show that the two configurations produce the same run. The crucial observation is that given any state of $C^{J,R}_{S,S}$, we can produce a run of $C^{J}_{S,S}$ which ends in the same state. In other words, our proof not only suggests that $C^{J,R}_{S,S}$ is equivalent to $C^{J}_{S,S}$ from a proof theoretic point of view, it also implies that the two configurations are equivalent for satisfiable formulas.

3.5.2 Equivalence between DPLL Solvers with ND Variable Selection on UNSAT Formulas

We show that when considered as a proof system, a DPLL solver with non-deterministic dynamic variable selection, arbitrary value selection and no restarts ($D^T_{ND,*}$) is p-equivalent to DPLL solver with non-deterministic dynamic variable and value selection and restarts ($D^{T,R}_{ND,ND}$), and hence, transitively p-equivalent to tree-like resolution—the restriction of general resolution where each consequent can be an antecedent in only one later inference.

Theorem 5. $D^T_{ND,*} \sim_p D^T_{ND,ND}$.

Proof. To show that $D^T_{ND,ND}$ p-simulates $D^T_{ND,*}$, we argue that every proof of $D^T_{ND,ND}$ can be converted to a proof of same size in $D^T_{ND,*}$. Let F be an unsatisfiable formula. Recall that a run of $D^T_{ND,ND}$ on F begins with non-deterministically picking some variable x to branch on, and a truth value to assign to x. W.l.o.g. suppose that the solver assigns x to 1. Thus, the solver will first refute $F[x = 1]$ before backtracking and refuting $F[x = 0]$.

To simulate a run of $D^T_{ND,ND}$ with $D^T_{ND,*}$, since variable selection is non-deterministic, $D^T_{ND,*}$ also chooses the variable x as the first variable to branch on. If the value selection returns $x = \alpha$ for $\alpha \in \{0, 1\}$, then the solver focus on the restricted formula $F[x = \alpha]$ first. Because there is no clause learning, whether $F[x = 1]$ or $F[x = 0]$ is searched first does not affect the size of the search space for the other. The proof follows by recursively calling $D^T_{ND,*}$ on $F[x = 1]$ and $F[x = 0]$. The converse direction follows since every run of $D^T_{ND,*}$ is a run of $D^T_{ND,ND}$. □

Corollary 2. $D^T_{ND,*} \sim_p D^{T,R}_{ND,ND}$.

Proof. This follows from the fact that $D^{T,R}_{ND,ND} \sim_p D^T_{ND,ND}$. Indeed, with non-deterministic branching and without clause learning, restarts cannot help. If ever $D^{T,R}_{ND,ND}$ queries a variable $x = \alpha$ for $\alpha \in \{0, 1\}$ and then later restarts to assign it to $1 - \alpha$, then $D^T_{ND,ND}$ ignores the part of the computation when $x = \alpha$ and instead immediately non-deterministically chooses $x = 1 - \alpha$. □

It is interesting to note that while the above result establishes a p-equivalence between DPLL solver models $D^T_{ND,*}$ and $D^{T,R}_{ND,ND}$, the following corollary implies that DPLL solvers with non-deterministic variable and randomized value selection are exponentially separable for satisfiable instances.

3.5.3 Separation Result for Drunk DPLL Solvers

We show that DPLL solvers with non-deterministic variable selection, randomized value selection and no restarts ($D^{T}_{ND,RD}$) is exponentially weaker than the same configuration with restarts ($D^{T,R}_{ND,RD}$).

Corollary 3. *$D^{T}_{ND,RD}$ runs exponentially slower on the class of satisfiable formulas $Ladder_n(G, f)$ than $D^{T,R}_{ND,RD}$, with high probability.*

The separation follows from the fact that our proof of the upper bound from Theorem 1 does not use the fact the solver has access to clause learning, which means the solver $D^{T,R}_{ND,RD}$ can also find a satisfying assignment for $Ladder_n(G, f)$ in time $O(n^2)$, except with exponentially small probability. On the other hand, the lower bound from Theorem 1 immediately implies an exponential lower bound for $D^{T}_{ND,RD}$, since $D^{T}_{ND,RD}$ is strictly weaker than $C^{T}_{ND,RD}$.

3.5.4 Separation Result for CDCL Solvers with WDLS

Finally, we state an observation of Robert Robere [99] on restarts in the context of the Weak Decision Learning Scheme (WDLS).

Definition 8 (WDLS)**.** *Upon deriving a conflict, a CDCL solver with WDLS learns a conflict clause which is the disjunction of the negation of the decision variables on the current assignment trail.*

Theorem 6. *$C^{J}_{ND,ND}$+WDLS is exponentially weaker than $C^{J,R}_{ND,ND}$+WDLS.*

Proof. The solver model $C^{J}_{ND,ND}$ with WDLS is only as powerful as $D^{T}_{ND,ND}$, since each learnt clause will only be used once for propagation after the solver backtracks immediately after learning the conlict clause, and remains satisfied for the rest of the solver run. This is exactly how $D^{T}_{ND,ND}$ behaves under the same circumstances. On the other hand, WDLS is an asserting learning scheme [92], and hence satisfies the conditions of the main theorem in [93], proving that CDCL with any asserting learning scheme and restarts p-simulates general resolution. Thus, we immediately have $C^{J,R}_{ND,ND}$ with WDLS is exponentially more powerful than the same solver but with no restarts (for unsatisfiable instances). □

3.6 Related Work

In this section we limit ourselves to previous work as it pertains to understanding the power of restarts, both from empirical and theoretical points of view. We refer the reader to Liang et al. [75] for a more detailed discussion of well-known restart heuristics such as Luby [77].

Previous Work on Theoretical Understanding of Restarts: Buss et al. [27] and Van Gelder [113] proposed two proof systems, namely regWRTI and pool resolution respectively, with the aim of explaining the power of restarts in CDCL solvers. Buss et al. proved that regWRTI is able to capture exactly the power of CDCL solvers with *non-greedy BCP* and without restarts and Van Gelder proved that pool resolution can simulate certain configurations of DPLL solvers with clause learning. As both pool resolution and regWRTI are strictly more powerful than regular resolution, a natural question is whether formulas that exponentially separate regular and general resolution can be used to prove lower bounds for pool resolution and regWRTI, thus transitively proving lower bounds for CDCL solvers without restarts. However, since Bonet et al. [25] and Buss and Kołodziejczyk [28] proved that all such candidates have short proofs in pool resolution and regWRTI, the question of whether CDCL solvers without restarts are as powerful as general resolution still remains open.

Previous Work on Empirical Understanding of Restarts: The first paper to discuss restarts in the context of DPLL SAT solvers was by Gomes and Selman [48]. They proposed an explanation for the power of restarts popularly referred to as "heavy-tailed explanation of restarts". Their explanation relies on the observation that the runtime of randomized DPLL SAT solvers on satisfiable instances, when invoked with different random seeds, exhibits a heavy-tailed distribution. This means that the probability of the solver exhibiting a long runtime on a given input and random seed is non-negligible. However, because of the heavy-tailed distribution of solver runtimes, it is likely that the solver may run quickly on the given input for a different random seed. This observation was the motivation for the original proposal of the restart heuristic in DPLL SAT solvers by Gomes and Selman [48].

Unfortunately, the heavy-tailed explanation of the power of restarts does not lift to the context of CDCL SAT solvers. The key reason is that, unlike DPLL solvers, CDCL solvers save solver state (e.g., learnt clauses and variable activities) across restart boundaries. Additionally, the efficacy of restarts has been observed for both deterministic and randomized CDCL solvers, while the heavy-tailed explanation inherently relies on randomness. Hence, newer explanations have been proposed and validated empirically on SAT competition benchmarks. Chief among them is the idea that "restarts compact the assignment trail

during its run and hence produce clauses with lower literal block distance (LBD), a key metric of quality of learnt clauses" [75].

Our Separation Results and Heavy-tailed Explanation of Restarts: A cursory glance at some of our separation results might lead one to believe that they are a complexity-theoretical analogue of the heavy-tailed explanation of the power of restarts, since our separation results are over randomized solver models. We argue this is not the case. First, notice that our main results are for drunk CDCL solvers that save solver state (e.g., learnt clauses) across restart boundaries, unlike the randomized DPLL solvers studied by Gomes et al. [48]. Second, we make no assumptions about independence (or lack thereof) of branching decisions across restarts boundaries. In point of fact, the variable selection in the CDCL model we use is non-deterministic. Only the value selection is randomized. More precisely, we have arrived at a separation result without relying on the assumptions made by the heavy-tailed distribution explanation, and interestingly we are able to prove that the "solver does get stuck in a bad part of the search space by making bad value selections". Note that in our model the solver is free to go back to "similar parts of the search space across restart boundaries". In fact, in our proof for CDCL with restarts, the solver chooses the same variable order across restart boundaries.

3.7 Conclusions

In this chapter, we prove a series of results that establish the power of restarts (or lack thereof) for several models of CDCL and DPLL solvers. We first showed that CDCL solvers with backtracking, non-deterministic dynamic variable selection, randomized dynamic value selection, and restarts are exponentially faster than the same model without restarts for a class of satisfiable instances. Second, we showed CDCL solvers with VSIDS variable selection and phase saving without restarts are exponentially weaker than the same solver with restarts, for a family of unsatisfiable formulas. Finally, we proved four additional smaller separation and equivalence results for various configurations of DPLL and CDCL solvers.

By contrast to previous attempts at a "theoretical understanding the power of restarts" that typically assumed that variable and value selection heuristics in solvers are non-deterministic, we chose to study randomized or real-world models of solvers (e.g., VSIDS branching with phase saving value selection). The choices we made enabled us to more effectively isolate the power of restarts in the solver models we considered. This leads us to the belief that the efficacy of restarts becomes apparent only when the solver models

considered have weak heuristics (e.g., randomized or real-world deterministic) as opposed to models that assume that all solver heuristics are non-deterministic.

Chapter 4

A Reinforcement Learning based Reset Policy for CDCL SAT Solvers

Restart policies are an important and widely studied class of techniques used in state-of-the-art Conflict-Driven Clause Learning (CDCL) Boolean SAT solvers, wherein some part of the state of solvers is erased at certain intervals during the run of the solver. In most modern solvers, variable activities are preserved across restart boundaries. An implication of this is that solvers continue to search parts of the assignment tree that are not far from the one immediately prior to a restart. To enable the solver to potentially search distant parts of the assignment tree, we study the effect of resets, a variant of restart which not only erases the assignment trail, but also randomizes the activity scores of the variables of the input formula, thus potentially enabling a better *global exploration* of the search space. The crucial question one needs to address here is when to invoke reset because randomizing variable activity can lower solver performance for certain classes of instances such as ones obtained from verification applications.

In this chapter, we model the problem of whether or not to trigger reset as a multi-armed bandit (MAB) problem, and propose a reinforcement learning (RL) based reset policy using Thompson sampling. The Thompson sampling algorithm is designed to balance the exploration-exploitation tradeoff by probabilistically and adaptively choosing arms (reset vs. no reset) based on their estimated rewards during the solver's run. The proposed reset policy is compared with state-of-the-art CDCL solvers Kissat, Kissat_MAB-HyWalk and MapleSAT on a set of 500 Satcoin benchmarks as well as 400 Main Track instances from the SAT competition 2022. Our results show that the proposed RL based reset policy convincingly outperforms their baseline solvers on Satcoin benchmarks, while retaining its competitiveness on the SAT competition instances, suggesting that the adaptive aspect of

our RL policy helps the solver to dynamically and profitably adapt the reset frequency for a given input instance.

4.1 Introduction

Over the last few decades, Conflict-Driven Clause Learning (CDCL) solvers have had a dramatic impact on many areas of software engineering [29], security [40, 117], and AI [24]. This is in part due to heuristics such as branching methods (e.g., VSIDS [83] and LRB [72]), deletion techniques [47], and restart policies [49], to name just a few. A fundamental problem that solver designers often face is that a set of techniques that work well for a class of instances can fail miserably for another.

An excellent example of this phenomena is the widespread use of restart policies such as Luby [77]. While these policies may differ in terms of frequency of restart, almost all of them have one feature in common, namely, that activities of variables are not changed across restart boundaries. There is good empirical reason for this design choice. It has been observed that setting activities to zero or randomizing the variable ranking in addition to everything that a traditional restart does (which we refer to as reset[1]), can somewhat negatively affect solver performance on SAT competition 2022 Main Track instances. On the other hand, in a recently published paper on restart policies [114, 70], it was mathematically shown that reset outperforms restart policies on a class of crafted instances called pitfall formulas.

It is therefore natural to ask "can we get the best of both worlds, i.e., develop an adaptive technique that dynamically and optimally switches between restarts and resets during the run of a solver for a given instance such that it outperforms, in terms of solving time, any other possible ordering of restart and reset calls?" In this chapter, we address this question from an empirical point of view.

To motivate the exploration of reset policies we start by making a few observation about traditional restarts methods. One well-known observation is that since activities are preserved across restart boundaries[2], the part of search space explored by the solver is not all that different from what it was searching prior to the restart. We refer to such behavior

[1]We do not claim to be the first to come up with the idea of reset. Instead, our work focuses on using reinforcement learning to optimally and adaptively switch between restart and reset policies for a given instance. We do introduce this new term reset, with the goal of clarifying the distinction between traditional restarts vs. resets.

[2]Informally, the term restart boundary refers to a point in time, measured from the start of a solver run, when a restart occurs.

of the solver as "local". Informally, we say that the solver's search is local, if the variable order defined by the activities of the variables after a restart is "similar" to the ones prior to that restart. We say that the solver's search is global, if the variable order across restart boundaries are "far apart".

This chapter is motivated by an observation we made about the performance some of the best SAT solvers, namely Kissat [42], Kissat_MAB-HyWalk [120] and MapleSAT [72], on cryptographic instances in general, and Satcoin instances in particular[3]. Specifically, we observed that both these solvers performed poorly on such instances. Upon experimenting with a variety of restart policies, we finally observed that when the activities were periodically reset to random values, the solvers performance improved dramatically. Further, as we increased the frequency of reset, we observed that these reset-based solvers performed better on Satcoin benchmarks. By contrast, these reset variants behaved poorly on SAT competition Main Track instances, even though their non-reset cousins performed well. Finally, as we experimented with greater frequency of reset, the performance of these reset variants got progressively worse.

This led us to ask the above-stated question for restarts vs resets, namely, "is there an adaptive technique that can switch between restarts and resets dynamically during a solver's run for a given instance?".

We address the above question by proposing a novel approach for CDCL solvers using a reinforcement learning reset policy, specifically with the Thompson sampling algorithm, modeling restarts and resets as an MAB problem with two arms. The decision between restart and reset is driven by a reward mechanism using learning rate (LR), an empirical measure of SAT solving efficiency. Our technique's efficacy is demonstrated through experiments on state-of-the-art solvers against their reset variants.

4.1.1 Contributions.

1. First, we model the problem of deciding whether to reset as a multi-armed bandit (MAB) problem, and we solve the problem using a method called Thompson sampling. In order to adapt the Thompson Sampling algorithm to the context of CDCL solvers, where the underlying search space changes may get restricted over time due to clause learning, and then expand due to clause deletion, we came up with a novel technique which decays the shape parameters used in the Thompson Sampling algorithm.

[3]These instances are obtained from a bitcoin mining application that uses the SHA256 Hash function [78].

2. Second, we also propose *partial resets*, a variant of resets: instead of randomly shuffling the activity scores for all variables, a partial reset preserves the order of top activity variables before the reset. That is, the top k after a reset is exactly the set of variables with top k activities in the exact same order. This technique aims at preserving some "locality" information of the branching heuristic while reinitializing the search to be "global" at the same time. We also observe that from a theoretical point of view, a partial reset is strictly weaker than a full reset. That is, there exists a class of formulas for which CDCL solver with VSIDS-like branching and full reset has a polynomial upper bound, whereas the same configuration with full reset replaced by partial reset has an exponential lower bound.

3. We empirically evaluated our RL reset policy on state-of-the-art solvers like Kissat, Kissat_MAB-HyWalk, and MapleSAT, using two benchmark sets. Our modified solvers significantly outperformed baselines on Satcoin instances, crypto benchmarks from bitcoin mining, solving all 500 instances in under 2000 seconds compared to the baseline's less than 15 instances within a 5000 second timeout. Despite this improvement on crypto benchmarks, our RL reset solvers maintained competitive performance on Main Track instances from the SAT competition 2022 and solved even more instances with our partial reset technique for baseline solvers MapleSAT and Kissat.

4.2 Reset Models

The problem of whether to restart or reset in a SAT solver can be modeled as a Reinforcement Learning (RL) problem by appropriately defining the states, actions, rewards, and the learning objective, similar to previous work on RL for SAT heuristics such as RL-based branching [72] as well as restarts [75].

1. **States:** The states represent the current state of the SAT solver, including the assignment of truth values to the variables, the current clause database, and any other relevant information.

2. **Actions:** The actions represent the decision of whether to perform a restart or a reset.

3. **Rewards:** The rewards represent the quality of the actions being taken. In our model, we use the local learning rate (llr) between two decision points to measure

the quality of the action. The agent is rewarded if llr is improved and penalized otherwise.

4. **Learning Objective:** The learning objective is to find a policy, that maximizes the expected llr over time.

The RL agent starts in some initial state, takes an action, observes the new state and reward, and updates its policy based on the observed experience. The agent should balance between the exploitation of the current knowledge and the exploration of new actions to learn better policies.

The problem of whether to restart or reset in a SAT solver is a good fit for RL because it is a sequential decision-making problem where the outcomes of actions are dependent on the current state of the environment. RL provides a way to learn a policy that maximizes the expected cumulative reward over time, which is particularly useful when the reward is uncertain or difficult to compute analytically.

We first give the definition of local learning rate(llr). "Local" here refers to the solver's performance (via the notion of learning rate) in between two consecutive restart boundaries. We then use llr to evaluate whether the previous pull of arms is a success or failure.

Definition 9 (local learning rate(llr))**.** *Let $t_1, t_2, t_3, ...t_k$ be timestamps of when restarts occur. We denote the number of learnt clauses added in between t_i and t_{i+1} by $num\_c(i, i+1)$, the number of decisions the solver made by $num\_d(i, i+1)$. We define the notion of local learning rate (llr), and denote the llr between t_i and t_{i+1} as $llr(i, i+1)$, where $llr(i, i+1) = \frac{num\_c(i,i+1)}{num\_d(i,i+1)}$.*

Traditionally, a range of metrics including solving time, number of decisions, number of conflicts, average width and many others have been used to evaluate SAT solver's performance. In this chapter we choose to use llr, offers a new and meaningful dimension for assessing SAT solver effectiveness. Different solvers have varied branching behaviours, learning techniques, clause deletion policies, making it challenging to compare them on a like-for-like basis. LLR gives a normalized measure of how effective a solver is at generating learned clauses for every decision it makes. If a solver learns more clauses with the same number of decisions, it means that the solver is more efficient in exploring its current search space. This efficiency could lead to faster solving time.

For the rest of this section, we describe five reset models we study in this chapter.

Full Reset with Fixed Probabilities In this model, we first predetermine a probability p. At restart boundaries, we generate a random number r, if $r \leq p$, then reset is performed, else reset is not performed. The goal of this model is to serve as a baseline model for exploring the potential of resets.

Decision based Full Reset with Thompson Sampling For this method, we adapt the Thompson sampling technique for solving our MAB model of the reset problem. We do this by first defining what is meant by success and failure. Ideally, we would like to count as success those actions that lead to a better search space where the solver can make progress faster. There have been various metrics proposed and studied to measure progress, including metrics to capture the quality of learnt clauses, decisions etc. For example, the size of clauses and literal block distance (LBD) [10] are well-known metrics to measure the quality of learnt clauses. Activities and global learning rate are examples of quality metrics for decisions [83, 76]. And lastly, learning rate and propagation rates are examples of metrics which try to capture the quality of the performance of the solver in the underlying search space [72, 87]. In this particular model, we choose a design which gives us a simplest model using Thompson sampling. Each time the solver makes a decision on a variable, we let the solver perform BCP till saturation, and if the solver hits a conflict, we consider the decision variable being good, and then increase the reward count for no restart by one, and if the decision does not lead to a conflict, we would increase the reward for restart by one. The intuition is that we would like to encourage a solver to restart if a sequence of decision does not lead to many conflicts.

Full Reset with Thompson Sampling For this model We chose the notion of learning rate to evaluate the performance of the arms in our model because it is well known to be an effective measure of solver's progress [72]. To be more specific, we use an *exponential moving average (EMA)* of local learning rates, which we refer to as EMA(llr), to keep track of each arm's historical performance, and then consider the pulling of an arm at t_i as a success if EMA(llr) $< llr(i, i+1)$. Otherwise when EMA(llr) $\geq llr(i, i+1)$, we would increase the failure count by one. Figure 4.1 contains the overall structure of reset model using Thompson Sampling.

Full Reset with Thompson Sampling and Decaying Shape Parameters In the context of whether to restart or reset, the output of the Thompson sampling algorithm relies on the historical performance of resetting. However during a solver's run, the solver keeps learning conflict clauses as well as deleting useless clauses over time, which means

Algorithm 1 RL reset with decaying shape parameters

```
if If restart criteria is met: then
    update_EMA_llr()
    if EMA_llr < llr: then
        last_arm.alpha += 1
    else
        last_arm.beta += 1
    end if
    decay(alpha1, beta1, alpha2, beta2)
    Erase the assignment trail
    last_arm = pick_arm(alpha1, beta1, alpha2, beta2)
    if last_arm == reset: then
        randomize_activity_scores()
    end if
end if
```

the underlying search space gradually changes all the time. Thus a success (or a failure) for an arm at the beginning of the run may be quite useless for determining which arm to pull at a later stage of the solve run. To solve this problem, we apply a decay, $0 < d < 1$, to the shape parameters every time one of the shape parameters gets updated, to give more weights to successes and failures of the arms' recent performances. Without loss of generality, assume at the time we would like to increase α by one, instead of just adding one to α, we do $\alpha_{new} = \alpha * d + 1$, and at the same time, we update β as well by doing $\beta_{new} = \beta * d$. See Algorithm 1 for pseudocode. Through our experiments, we found that decaying shape parameters are crucial to the performance of our RL reset policy.

Partial resets We also explore a variant of full reset referred to as partial reset. As defined earlier, in a partial reset the activities of the top-k variables are retained across reset boundaries and the activities of the remaining variables are randomized. To achieve this, the ordering of top k variables are recorded before randomizing activities, and after randomizing activities for all variables, the top k variables receive a constant bump, just enough so that they still have top activities, and in the same order as before the reset. The reason for exploring partial resets is that full reset may not be the best strategy for industrial instances that have a lot of "locality".

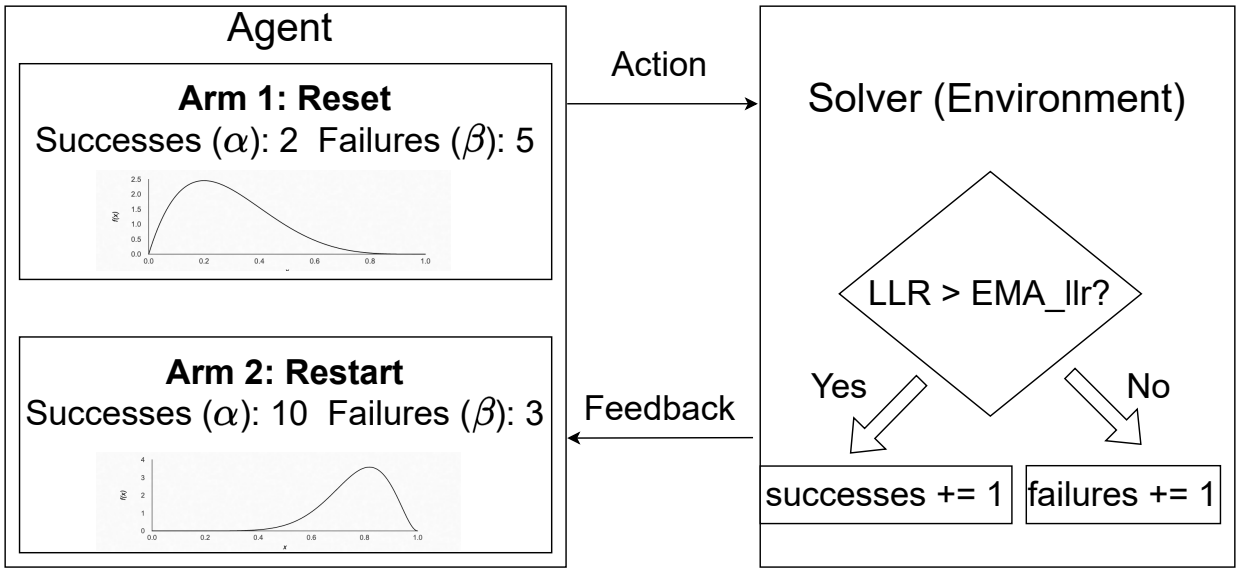

Figure 4.1: The figure describes our reset method. Immediately prior to calling reset (and after the assignment trail has been deleted), we sample from the beta distribution for each arm and pick the arm with the higher of the two sampled values. Further, we then compute the llr after the action is taken, update the EMA_llr, and compare the llr and EMA_llr. If llr $>$ EMA_llr, the number of success is increased by 1 for the selected arm, and if llr $\leq$ EMA_llr, the number of failure is increased by 1.

4.3 Theoretical Observations

4.3.1 Lower Bound for Partial Reset

We observe that solvers using a particular partial reset model suffer from the same exponential lower bound as those using VSIDS-like branching heuristic and restart as shown by Viyals [114]. However, in a separate paper, Li et al. [70] showed that these same set of instances can be solved in polynomial time by solvers with VSIDS-like branching and full reset.

All statements below are probabilistic; the probability comes from the fact that we are resolving ties between variables with the same score at random. We first state the relevant theorems.

Theorem 7. *[114][Vinyal's Theorem] There is a family of instances, called pitfall formulas, that have polynomial resolution proofs, but CDCL with VSIDS and restarts requires exponential time to solve them, except with exponentially small probability.*

Theorem 8. *[70] The family of formulas from Theorem 7 can be solved in polynomial time by CDCL with VSIDS and full resets, except with exponentially small probability.*

Observation 1. *Consider any partial reset strategy as defined below: the activities of variables that have positive (respectively, zero) activity score prior to a reset remain positive (respectively, zero) after a reset. Given the lower bound theorem by Vinyals 7, we can easily see that CDCL with VSIDS and a partial reset strategy requires exponential time to solve the pitfall formulas, except with exponentially small probability.*

To see this, first recall Vinyal's theorem applies to any solver that satisfies Definition 3.1 as defined in his paper [114]. Briefly, Definition 3.1 roughly states that VSIDS-like branching heuristics always prefer variables that have participated in a conflict over variables that have never been assigned.

Observe that CDCL with VSIDS-like branching and partial reset satisfies this definition since any variable that participates in a conflict has a positive activity and those which have not been assigned have a 0 activity score. After a partial reset, these scores are carried over. Given the structure of pitfall formulas, this behavior of reset implies that if the solver is starts branching in the hard component of the pitfall formulas, then it cannot escape this hard part without actually constructing an exponential-sized proof.

Further note that, in their paper Li et al. [70] showed that if restarts are replaced with full reset in a CDCL solver with VSIDS-like branching, then it has a polynomial upper bound for pitfall formulas. Thus, CDCL solvers with VSIDS-like branching and partial reset can be exponentially weaker than ones with full reset.

4.3.2 Bounds with Decaying Shape Parameters

One reason to apply a decay to the shape parameters used in Thompson Sampling is to adapt to the change in the underlying search space. Here, we would like to argue from another angle that the decay is needed.

First, note that the values of the shape parameters are upper-bounded by the sum of the geometric series $lim_{k\to\infty} 1 + d + d^2 + d^3 ... = \frac{1}{1-d}$, which is a constant. Therefore, for small values of d, the shape parameters stay at low values at all times.

Also recall the formulas for the mean and variance for the beta distribution: if $X \sim \text{beta}(\alpha, \beta)$, then the mean of X is

$$E[X] = \frac{\alpha}{\alpha + \beta}$$

and the variance of X is

$$Var[X] = \frac{\alpha\beta}{(\alpha + \beta)^2(\alpha + \beta + 1)}$$

Without a decay, the shape parameters α and β may get arbitrarily large and become unbounded. When shape parameters for Thompson Sampling are large and unbounded, the variance when sampling from the underlying beta distribution becomes very small, leading to the following simple observation.

Observation 2. $\lim_{\alpha\to\infty} Var[X] \to 0$ *and* $\lim_{\beta\to\infty} Var[X] \to 0$.

This means that the solver would keep picking the same arm, even if the arm fails to perform. A constant change in the values of the shape parameters would not change the underlying distribution too much, the mean would not change much and the variance would remain low, leading to the following observation. Thus, the Thompson Sampling algorithm becomes less exploratory, at least for very long time intervals.

Observation 3. *When* $\alpha >> \beta$ *and* $\alpha \to \infty$, $\frac{\alpha}{\alpha+\beta}$ - $\frac{\alpha}{\alpha+\beta+1} \to 0$.

On the other hand, when we apply a decay to the shape parameters, the shape parameters are upper bounded by a constant. Applying a constant change in the values of the shape parameters would change the mean by at least a constant, thus making the algorithm more exploratory, leading to the following observation.

Observation 4. *For* $\alpha, \beta < c$ *where* c *is a constant,* $\frac{\alpha}{\alpha+\beta}$ - $\frac{\alpha}{\alpha+\beta+1} > \frac{1}{4c+2}$.

4.4 Experimental Results

In this Section, we report on the extensive experimental evaluation we performed to test the above-described full and partial reset policies as implemented in Kissat, Kissat_MAB-HyWalk and MapleSAT solvers.

	Number of solved instances	
	SATComp 2022	Satcoin
K	272	0
K_RLDecay_full	276	360
K_RLDecay_partial	274	496
Khy	**289**	7
Khy_RLDecay_full	271	**500**
Khy_RLDecay_partial	277	489
M	178	13
M_RLDecay_full	178	**500**
M_RLDecay_partial	182	**500**

Table 4.1: Number of solved instances for baseline solvers Kissat(K), Kissat_MAB-HyWalk(Khy) and MapleSAT(M) along with their reset variants. Observe that Kissat, Kissat_MAB-HyWalk and MapleSAT perform poorly on Satcoin instances. By contrast, the reset variants of Kissat and MapleSAT outperform their respective baselines on Main Track 2022 instances, and vastly outperform on Satcoin instances together with Kissat_MAB-HyWalk's reset variants.

4.4.1 Benchmarks

1. **Satcoin instances [78]:** These instances are derived from Bitcoin mining problems. Satcoin instances are search problems that involve finding a nonce value that, when combined with a block header, results in a hash value that is lower than a given difficulty target. By varying the difficulty target, one can generate Satcoin instances of various hardness and perform scaling studies. These instances are highly challenging and require a significant amount of computational effort to solve.

2. **Main Track instances from SAT Competition 2022 [102]:** The Main Track instances are a set of problems used in SAT Competition 2022. The Main Track instances are designed to be representative of real-world SAT problems and cover a wide range of applications, including verification, planning, and scheduling. The instances are a mix of easy, medium, and hard instances. The Main Track instances are a widely recognized benchmark for evaluating the performance of SAT solvers.

	PAR2 scores	
	maintrack2022	Satcoin
K	3685.04	9749.96
K_RLDecay_full	3611.54	4023.91
K_RLDecay_partial	3691.41	569.81
Khy	**3345.14**	9841.02
Khy_RLDecay_full	3802.69	452.16
Khy_RLDecay_partial	3680.96	722.37
M	5246.65	9744.82
M_RLDecay_full	5258.53	**403.28**
M_RLDecay_partial	5689.92	463.19

Table 4.2: PAR2 scores for baseline solvers Kissat, Kissat_MAB-HyWalk and MapleSAT along with their reset variants.

4.4.2 Experimental Setup

We ran each solver configuration on each instance on Intel E5-2683 v4 Broadwell @ 2.1GHz CPUs running Linux 3.10.0-1160.88.1.el7.x86 64 (Digital Research Alliance of Canada), with a timeout of 5000 seconds of CPU time. The performance of the solvers is compared using PAR-2 scores. Parameters and metrics being used are standard in the SAT community [102].

Since the baseline solvers all have an option to enable randomness, we use the same default random seeds as the baseline solvers. We use 0.8 for both computing the EMA_llr as well as decaying the shape parameters used in Thompson sampling.

4.4.3 Full Reset with Fixed Probabilities

For a fixed probability reset, we pre-determine a probability p and we perform a reset with probability p at restart boundaries. We make the following observations:

1. With the fixed probability resetting policy, we solve significantly more Satcoin instances than the baseline solver, which can barely solve any of these instances. Even with 5% probability of reset, the reset solver solves more than 200 out of 500 Satcoin instances within 5000 seconds timeout limit.

2. There is a positive correlation between the reset probability and the performance of the solver on Satcoin instances. As we increase the probability, the reset solvers

solve all 500 out of 500 Satcoin instances at 20% fixed probability reset. When the probability goes up from 20% to 50%, the time used to solve all 500 instances continues to reduce.

However, by using fixed probability reset policy on Main Track instances from SAT competition 2022, completely different observations are made:

1. When we have a relatively small reset probability, the reset policy's performance can match the traditional CDCL solver. For example, 5% reset probability MapleSAT solver can solve 178 instances, which is the same number of solved instances by the base solver.

2. As we increase the reset probability, the performance on Main Track instances degrades significantly. To give an example, 50% reset probability MapleSAT solver solves 166 instance while the base solver solves 178 instances.

The experiments with fixed probability reset policy demonstrate its advantages in solving Satcoin instances, and a higher reset probability is desired to solve Satcoin instances efficiently. Nonetheless, an overly resetting solver leads to poor performance on Main Track instances.

4.4.4 Full Reset with and without Thompson Sampling and Decaying Shape Parameters

In our experiments, we noticed that the Thompson sampling algorithm was significantly influenced by the initial warm-up phase of the solvers, leading to a bias against resets. This was because the early performance metrics didn't accurately represent the search space. To adapt the constantly changing search space, we developed RL reset solvers with decaying alpha and beta values. The results are as follows:

Decaying Shape Parameters

1. For Satcoin instances, solvers with RL reset policies without decaying alpha and beta values barely solves any instances. This is due to the warm-up bias mentioned above. In the early stage of the search, the RL model does not really trigger a reset. This bias was taken to the later stage of the search, which significantly reduces the number of resets being carried out. As we discussed in the experiments with fixed reset

probabilities, Satcoin instances can only be solved efficiently with higher probability of reset. With decaying alpha and beta values, the RL model focuses more on the outcomes of recent decisions, thus being more exploratory, and more resets are carried out even when the recent search does not produce good performance according to llr. For RL reset with decaying alpha and beta values, we not only solve all 500 Satcoin instances, but also perform almost as efficiently as the 50% fixed probability reset solver from previous experiments, except much resets needed. Please see the Appendix for a more detailed analysis.

2. For Main Track instances, the RL reset with decaying alpha and beta values slightly improved based solvers' performance for Kissat and MapleSAT. However, for Kissat_MAB-HyWalk, the baseling solver remains the top.

After analyzing the experimental results for RL reset, it is clear that the RL reset reduces the performance loss of the solver over Main Track instances, while preserving its power in Satcoin formulas by dynamically adjusting the likelihood of triggering a reset.

Decision based and LLR based resets

We observe that for decision based reset policy which modifies reward for reset and no reset every decision point, the performance of the underlying solver is worse than the model where we update the reward after a window of conflict (See Figure 4.2). Our hypothesis is that, updating per decision can introduce bias. Simply rewarding restart when a sequence of decisions do not lead to conflicts may not be beneficial as certain search space or conflicts relies on having multiple assumptions being made and requres a longer trail, and it is possible that after a long trail the solver start making progress. However when we consider LLR based techniques, we evaluate solver performance for a short history, and aggregate the progress the solver made through learning rate, which is a more gentle way of measuring progress.

4.4.5 Partial Reset

We observe that when implementing a partial reset strategy, there does not seem to be a obvious trend in how many of the top k variables should be preserved. For main track instances, among $k = 5, 10, 20$ and 30, $k = 5$ has the best performance, but yet $k = 10$ performs worse than $k = 30$. Additionally $k = 5$ not only performed better than other values of k, it also improved upon the performance of the baseline solver for MapleSAT and remains competitive for other baseline solvers. And for SATcoin instance, having partial

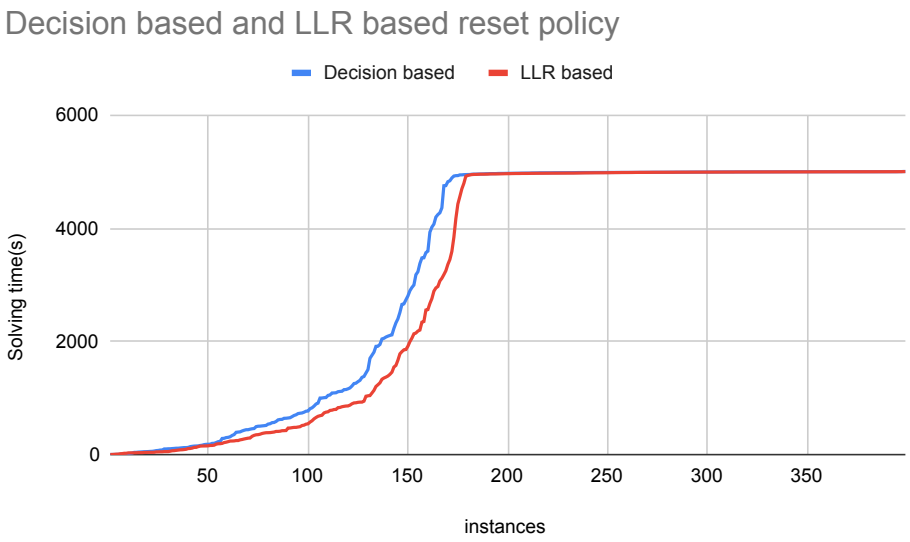

Figure 4.2: Solving times for MapleSAT with decision based reset policy vs LLR based reset policy over SATcomp benchmarks. LLR based reset policy has a clear advantage over decision based models.

restart improve the performance of Kissat with RL reset policy, and remains competitive for other baseline solvers.

4.5 Related Work

There has been considerable theoretical and empirical work on restarts. Liang et al. [75] provide a nice overview of the empirical work on restarts, while the paper by Li et al. [70] gives a thorough overview of the proof complexity of restarts. Below we focus mostly on reset policies, and contrast with restart methods if necessary.

4.5.1 Empirical Work on Reset Policies

Chaff [83] was the first CDCL solver to implement restart. CaDiCal [16] being the first CDCL solver with the idea of rephasing, where the phase value of variables is reset on certain intervals during the run of the solver. While many solver developers have experimented with reset policies, we are not aware of a thorough and systematic study such

techniques. To the extent that we know, we are the first to explore RL-based full and partial reset policies.

4.5.2 Theoretical Work on Reset Policies

The first paper to discuss restarts in the context of DPLL SAT solvers was by Gomes and Selman [48]. They proposed an explanation for the power of restarts popularly referred to as "heavy-tailed explanation of restarts". Their explanation relies on the observation that the runtime of randomized DPLL SAT solvers on satisfiable instances, when invoked with different random seeds, exhibits a heavy-tailed distribution. This means that the probability of the solver exhibiting a long runtime on a given input and random seed is non-negligible. However, because of the heavy-tailed distribution of solver runtimes, it is likely that the solver may run quickly on the given input for a different random seed. This observation was the motivation for the original proposal of the restart heuristic in DPLL SAT solvers by Gomes and Selman [48].

4.5.3 Use of Reinforcement Learning in Solvers

The use of RL techniques for switching between solver heuristics has been explored in the past. The earliest work for using RL to switch between heuristics that we are aware of is by Lougdakis et al. [65]. In their work, they switch between different branching heuristics. Liang et al. [72] were among the first to model branching as an MAB problem. Liang et al. [75] also developed an RL technique for switching between restart policies. Another example of the use of RL techniques to switch between branching heuristics at each restart is [31].

Our work differs from these previous methods in the following important ways. First, we adapt Thompson sampling to our setting to solve the problem of switching between reset heuristics. Second, we use a shaping mechanism to help our learning agent to explore more than it otherwise would. Third, we develop our technique for the setting of reset policies. Another point of differentiation is that we are not aware of previous work that thoroughly and systematically explores a variety of reset policies in multiple SAT solvers over a comprehensive benchmark.

4.6 Conclusion

In this chapter, we revisit the idea of resetting variable activities at certain intervals during the run of the solver. Specifically, we study a range of reset policies and devise a RL based technique to switch between restart and reset in the Kissat, Kissat_MAB-HyWalk and MapleSAT solvers. We observe that a full reset (i.e., randomizing the activity of all variables) performs dramatically well for Satcoin instances, irrespective of the baseline solver used. However, the same technique is somewhat weaker on the SAT competition 2022 main track instances compared to the corresponding baselines. This behavior of reset motivates the design of a reinforcement learning policy that adaptively chooses to invoke a reset based on the success of previous invocations as measured by an EMA over a metric we call local learning rate.

We implement our techniques in two state-of-the-art baseline solvers, namely, Kissat, Kissat_MAB-HyWalk and MapleSAT. Via extensive experimentation, we show that both our full and partial reset techniques vastly outperform the baseline on Satcoin benchmarks, while the partial reset solvers outperform the respective baselines on the SAT competition 2022 Main Track instances. We also show that by performing partial reset we improve the performance of the resetting solvers relative to full reset solvers, while maintaining exceptional performance over Satcoin instances.

Chapter 5

On the Hierarchical Community Structure of Practical Boolean Formulas

Modern CDCL SAT solvers easily solve industrial instances containing tens of millions of variables and clauses in them, despite the theoretical intractability of the SAT problem. This gap between practice and theory is a central problem in solver research. It is believed that SAT solvers exploit structure inherent in industrial instances, and hence there have been numerous attempts over the last 25 years at characterizing this structure via parameters. These can be classified as *rigorous*, i.e., they serve as a basis for complexity-theoretic upper bounds (e.g., backdoors), or *correlative*, i.e., they correlate well with solver run time and are observed in industrial instances (e.g., community structure). Unfortunately, no parameter proposed to date has been shown to be both strongly correlative and rigorous over a large fraction of industrial instances.

Given the sheer difficulty of the problem, we aim for an intermediate goal of proposing a set of parameters that is strongly correlative and has good theoretical properties. Specifically, we propose parameters based on a graph partitioning called Hierarchical Community Structure (HCS), which captures the recursive community structure of a graph of a Boolean formula. We show that HCS parameters are strongly correlative with solver run time using an Empirical Hardness Model, and further build a classifier based on HCS parameters that distinguishes between easy industrial and hard random/crafted instances with very high accuracy. We further strengthen our hypotheses via scaling studies. On the theoretical side, we show that counterexamples which plagued flat community structure do not apply

to HCS, and that there is a subset of HCS parameters such that restricting them limits the size of embeddable expanders.

5.1 Introduction

Over the last two decades, Conflict-Driven Clause-Learning (CDCL) SAT solvers have had a dramatic impact on software engineering [29], formal methods [32], security [40, 117], and AI [24], thanks to their ability to solve large real-world instances with tens of millions of variables and clauses [102], notwithstanding the fact that the Boolean satisfiability (SAT) problem is known to be NP-complete and is believed to be intractable [35]. A plausible explanation of this apparent contradiction would be that NP-completeness of the SAT problem is established in a worst-case setting, while the dramatic efficiency of modern SAT solvers is witnessed over "practical" instances. However, despite over two decades of effort, we still do not have an appropriate mathematical characterization of practical instances (or a suitable subset thereof) and attendant complexity-theoretic upper and lower bounds. This gap between theory and practice is rightly considered one of the central problems in solver research by theorists and practitioners alike.

The fundamental premise in this line of work is that SAT solvers are able to find short proofs (if such proofs exist) in polynomial time (i.e., they are efficient) for industrial instances and that they are able to do so because they somehow exploit the underlying properties (a.k.a. structure) of such industrial Boolean formulas[1], and, further, that hard randomly-generated or crafted instances are difficult because they do not possess such structure. Consequently, considerable work has been done in characterizing the structure of industrial instances via parameters. The parameters discussed in literature so far can be broadly classified into two categories: correlative and rigorous[2]. The term *correlative* refers to parameters that take a specific range of values in industrial instances (as opposed to random/crafted) and further have been shown to correlate well with solver run time. This suggests that the structure captured by such parameters might explain why solvers are efficient. An example of such a parameter is modularity (more generally community structure [6]). By contrast, the term *rigorous* refers to parameters that characterize classes of formulas that are fixed-parameter tractable (FPT), such as backdoors [116, 122], backbones [82], treewidth, and branchwidth [3, 101], among many others [101], or have been

[1]The term industrial is loosely defined to encompass instances obtained from hardware and software testing, analysis, and verification applications.

[2]Using terminology by Stefan Szeider [109].

used to prove complexity-theoretic bounds over randomly-generated classes of formulas such as clause-variable ratio (a.k.a., density) [34, 103].

The eventual goal in this context is to discover a parameter or set of parameters that is both strongly correlative and rigorous, such that it can then be used to establish parameterized complexity-theoretic bounds on an appropriate mathematical abstraction of CDCL SAT solvers, thus finally settling this decades-long open question. Unfortunately, the problem with all the previously proposed rigorous parameters is that either "good" ranges of values for these parameters are not witnessed in industrial instances (e.g., such instances can have both large and small backdoors) or they do not correlate well with solver run time (e.g., many industrial instances have large treewidth and yet are easy to solve, and treewidth alone does not correlate well with solving time [81]).

Consequently, many attempts have been made at discovering correlative parameters that could form the basis of rigorous analysis [6, 46]. Unfortunately, all such correlative parameters either seem to be difficult to work with theoretically (e.g., fractal dimension [4]) or have obvious counterexamples, i.e., it is easy to show the existence of formulas that simultaneously have "good" parameter values and are provably hard-to-solve. For example, it was shown that industrial instances have high modularity, i.e., supposedly good community structure [6], and that there is good-to-strong correlation between modularity and solver run time [89]. However, Mull et al. [84] later exhibited a family of formulas that have high modularity and require exponential-sized proofs to refute. Finally, this line of research suffers from important methodological issues, that is, experimental methods and evidence provided for correlative parameters tend not to be consistent across different papers in the literature.

Hierarchical Community Structure of Boolean Formulas: Given the sheer difficulty of the problem, we aim for an intermediate goal of proposing a set of parameters that is strongly correlative and has good theoretical properties. Specifically, we propose a set of parameters based on a graph-theoretic structure called Hierarchical Community Structure (HCS), inspired by a commonly-studied concept in the context of hierarchical networks [33, 97], which satisfies all the empirical tests hinted above and has better theoretical properties than previously proposed correlative parameters. The intuition behind HCS is that it neatly captures the structure present in human-developed systems which tend to be modular and hierarchical [106], and we expect this structure to be inherited by Boolean formulas modelling these systems.

Contributions[3]**:**

[3]Instance generator and data can be found at https://satsolvercomplexity.github.io/hcs. Also, for the full-length paper and appendices (with proofs of theorems in Section 5.5), please refer to the arXiv

1. **Empirical Result 1 (HCS and Industrial Instances):** We show that a set of parameters based on the HCS of the variable-incidence graph (VIG) of Boolean formulas are effective in distinguishing industrial instances from random/crafted ones. Moreover, we build a classifier that robustly classifies SAT instances into the categories they belong to (verification, random, etc.). The classification accuracy is approximately 99% and we perform a variety of tests to ensure there is no overfitting (See Section 5.4.1).

2. **Empirical Result 2 (Correlation between HCS and Solver Run Time):** We build an empirical hardness model based on our HCS parameters to predict the solver run time for a given problem instance. Our model, based on regression, performs well, achieving an R^2 score of 0.83, much stronger than previous such results (See Section 5.4.2)

3. **Empirical Result 3 (Scaling Experiments of HCS Instances):** We empirically show, via scaling experiments, that HCS parameters such as community degree and leaf-community size positively correlate with solving time. We empirically demonstrate that formulas whose HCS decompositions fall in a good range of parameter values are easier to solve than instances with a bad range of HCS parameter values (See Section 5.4.4).

4. **Theoretical Results:** We theoretically justify our choice of HCS by showing that it behaves better than other parameters. More concretely, we show the advantages of hierarchical over flat community structure by identifying HCS parameters which let us avoid hard formulas that can be used as counterexamples to community structure [84], and by showing graphs where HCS can find the proper communities where flat modularity cannot. We also show that there is a subset of HCS parameters (leaf-community size, community degree, and fraction of inter-community edges) such that restricting them limits the size of embeddable expanders (See Section 5.5).

5. **Instance Generator:** Finally, we provide an HCS-based instance generator which takes input values of our proposed parameters and outputs a formula that satisfies those values. This generator can be used to generate "easy" and "hard" formulas with different hierarchical structures (See Section 5.4.4).

Research Methodology: We also codify a set of empirical tests which we believe parameters must pass in order to be considered for further theoretical analysis. While other

version of the paper [68].

researchers have considered one or more of these tests, we bring them together into a coherent and sound research methodology that can be used for future research in formula parameterization (See Section 5.2). We believe that the combination of these tests provides a strong basis for a correlative parameter to be considered worthy of further analysis.

5.2 Research Methodology

As stated above, the eventual goal of the research presented here is to discover a structure and an associated parameterization that is highly correlative with solver run time, is witnessed in industrial instances, and is rigorous, i.e., forms the basis for an upper bound on the parameterized complexity [101] of the CDCL algorithm. Considerable work has already been done in attempting to identify exactly such a set of parameters [89]. However, we observed that there is a wide diversity of research methodologies adopted by researchers in the past. We bring together the best lessons learned into what we believe to be a sound, coherent, and comprehensive research methodology explained below. We argue that every set of parameters must meet the following empirical requirements in order to be considered correlative:

1. **Structure of Industrial vs. Random/Crafted Instances:** A requisite for a structure to be considered correlative is that industrial instances must fall within a certain range of values for the associated parameters, while random and crafted instances must have a different range. An example of such a structure is the community structure of the VIG of Boolean formulas, as parameterized by modularity. Multiple experiments have shown that industrial instances have high modularity (close to 1), while random instances tend to have low modularity (close to 0) [89]. This could be demonstrated via a correlation experiment or by building a classifier that takes parameter values as input features.

2. **Correlation between Structure and Solver Run Time:** Another requirement is correlation between parameters of a structure and solver run time. Once again, community structure (and the associated modularity parameter) forms a good example of a structure that passes this essential test. For example, it has been shown that the modularity of the community structure of industrial instances (resp. random instances) correlates well with low (resp. high) solver run time [89]. One may use either correlation methods or suitable machine learning predictors (e.g., random forest) as evidence here.

3. **Scaling Studies:** To further strengthen the experimental evidence, we require that the chosen structure and its associated parameters must pass an appropriately designed scaling study. The idea here is to vary one parameter value while keeping as much of the rest of the formula structure constant as possible, and see its effect on solver run time. An example of such a study is the work of Zulkoski et al. [121], who showed that increasing the mergeability metric has a significant effect on solver run time.

Limitations of Empirical Conclusions: As the reader is well aware, any attempt at empirically discovering a suitable structure (and associated parameterization) of Boolean formulas and experimentally explaining the power of solvers is fraught with peril, since all such experiments involve pragmatic design decisions (e.g., which solver was used, choice of benchmarks, etc.) and hence may lead to contingent or non-generalizable conclusions. For example, one can never quite eliminate a parameter from further theoretical analysis based on empirical tests alone, for the parameter may fail an empirical test on account of benchmarks considered or other contingencies. Another well-understood issue with conclusions based on empirical analysis alone is that they by themselves cannot imply provable statements about asymptotic behavior of algorithms. However, one can use empirical analysis to check or expose gaps between the behavior of an algorithm and the tightness of asymptotic statements (e.g., the gap between efficient typical-case behavior vs. loose worst-case statements). Having said all this, we believe that the above methodology is a bare minimum that a set of parameters must pass before being considered worthy of further theoretical analysis. In Section 5.4, we go into further detail about how we protect against certain contingent experimental conclusions.

Limits of Theoretical Analysis: Another important aspect to bear in mind is that it is unlikely any small set of parameters can cleanly separate all easy instances from hard ones. At best, our expectation is that we can characterize a large subset of easy real-world instances via the parameters presented here, and thus take a step towards settling the central question of solver research.

5.2.1 What is evidence?

In general, evidence is the result of testing a hypothesis. The precise definition of what constitutes evidence changes depending on the hypothesis that we want to test. In this work, our hypothesis is that our proposed set of parameters is predictive of the difficulty of SAT instances. For a predictive parameterization of the difficulty of SAT instances, a

formula is easy if and only if it has 'good' parameter values; conversely, a formula is hard if and only if it has 'bad' parameter values. This hypothesis is general and can be applied to any proposed set of parameters.

With this hypothesis, it is clear that any potential evidence which tests this hypothesis must relate parameter values to the solving times of CNF formulas. This definition includes empirical data on the parameter values and solving times of a CDCL solver on CNF formulas, instance generators and results derived from the analysis of this data, and theoretical results based on models of the system. This is the type of data that our proposed tests will produce. Analysis of this evidence should allow us to determine whether our parameters meet the criteria to be considered a predictive set of parameters. The evidence that we present in this work consists of two machine learning classifiers, scaling studies demonstrating the correlation between parameter values and solving time, and theoretical results identifying the relationships between our parameters and the size of the CDCL proof.

Description of Parameter Computation Apart from the running time and satisfiability data which we compute using a CDCL SAT solver, we also need to compute the values of the parameters of interest. We implemented a tool to compute the hierarchical community structure parameters of a formula. Our tool recursively computes the community structure of the formula using the Louvain method [23], producing a hierarchical tree by repeatedly decomposing communities into smaller sub-communities. Logging the values of the metrics of interest at every node in the hierarchy separates the processes of data collection and analysis, allowing us to later analyze the data without needing to aggregate all of the metrics immediately. The hierarchical structure of each formula is later recovered by parsing the stored data, and higher-level parameters are computed by iterating over the communities of the reconstructed HCS tree. We run all of our computing jobs on SHARCNET.

Our hierarchical parameter computations result in parameter data for each of the communities and sub-communities in the formula. In order to extract information from our computed data, we aggregate parameter values within each formula to provide a further abstraction of the properties of the formula. This involves computing average values for the parameters over every community as well as at each depth of the hierarchical tree. These aggregated parameters for each formula serve as the basis for further analysis.

5.2.2 How is evidence analyzed?

Evidence is analyzed by systematically looking for correlations in the data. This involves producing data visualizations in the form of various plots and histograms to develop intuition about the problem.

When testing whether different instance classes have different parameter values, we analyze the data in two ways:

1. Average value plots (parameter value vs. depth): we aggregate parameter values by computing averages over the instances belonging to each class and plotting these average parameter values for each depth. These plots give us an initial impression of whether the parameter values are different for different instance classes. We can then precisely quantify the difference between the classes by developing machine learning models.

2. Machine learning models: we train a series of machine learning classifiers to distinguish between different instance classes using our parameters as inputs. We randomly divide our benchmark instances into training and verification sets at a proportion of 80% training and 20% verification. To ensure the robustness of our results, we repeat this procedure five times and report the average classifier accuracies.

When testing whether a parameter correlates with running time, we analyze the data in two ways (similarly to when we are testing whether parameter values can be used to separate instance classes):

1. Scatter plots (parameter value vs. running time): for each instance, plot the parameter value against the solving time for the instance. We also consider log-log plots. Then, fit a trendline to the data and calculate the R^2 correlation value for the dataset. We measure the strength of the correlation by how close $|R^2|$ is to 1. In order to check whether combinations of our parameters might correlate with running time, we additionally consider linear combinations of our parameters. This process of combining parameters is similar to the technique used to test the combinations of parameters in [121]. In our work, we treat the timeout as the actual solving time of the instance. This can affect the quality of our correlation results.

2. Machine learning models: using our parameters, we train ML models to classify the formula into one of two classes: 'easy' or 'hard'. This technique is not affected by the fact that the true solving time is not represented for instances which time out. Regardless of the true running time, instances which time out are 'hard'.

- For the instance family classifier: Instances are labelled with the class of instances they were acquired from
- For the hardness classifier: Instances are labelled as 'easy' if their running times are less than the median running time and 'hard' otherwise

5.2.3 Caveats when analyzing evidence

In our analysis, we aggregate the collected data by computing averages. Within a formula, we argue that this is reasonable when the hierarchical tree is relatively balanced, i.e., the parameters do not take on extremely different values for different communities and branches of the hierarchical tree. Similarly, within a class of instances, simply taking averages is reasonable when the sample instances are all similar to each other. It is possible that a more sophisticated method exists for aggregating our data, but we are unaware of any such scheme which can be demonstrably justified for our data.

When drawing (contingent) conclusions after analyzing the data, it is imperative that we are aware of the assumptions that our conclusions rely upon. We note that regardless of how we define our parameters, there will always be other parameters which will be more predictive than our parameters in certain scenarios.

5.2.4 When is evidence considered to support a hypothesis?

In general, evidence is considered to support a hypothesis if the data does not contradict any of the predictions made by the hypothesis. Otherwise, the evidence is considered to refute the hypothesis. This means that there is a different way for each type of evidence to support or refute a hypothesis, depending on the different predictions made by the hypothesis. Our hypothesis is that a set of parameters is 'predictive', which is an imprecise statement and therefore hard to verify. Even with our four empirical requirements, In order to test the hypothesis, it must be refined further.

Ideally, the set of parameters we pick would be 100% accurate in meeting both of the criteria identified in subsection 5.2.1. However, in practice, it is highly unlikely that such a parameterization will ever be identified. Thus, we do not argue that the parameters we present are the ideal set of parameters, but simply that they are a *good* set of parameters. With this aim, we define a minimum threshold of accuracy/confidence that the results must meet before we consider the evidence to support the hypothesis.

For the machine learning models we present, we say that the evidence supports our hypothesis when the classification accuracies are high (above 75%). Otherwise, we say that the evidence refutes our hypothesis.

This chapter is founded on a series of empirical results where a SAT solver is used to compute solving times for an array of propositional formulas. Thus, our results are subject to the limitations and practicalities of the real world, and there are a vast number of design choices which must be made in order to produce a result. Some of the choices we make are inherently arbitrary, and in many cases, it is difficult to argue whether one decision or definition is better than another.

The validity of the conclusions in any empirical work is contingent on the soundness of these decisions. In general, empirical work is fraught with 'contingent conclusions' and practical decisions that may lead us astray. Hence, we are very cautious in making highly conservative conclusions. We can identify correlations, but we cannot determine cause-and-effect. This means that we will never have a sense of finality about when to stop looking at parameters, and that we cannot conclusively say that the parameters we favour in this chapter will stand the test of time. Given all of these caveats, all we can say is that our current crop of parameters seem better suited as a basis for further theoretical analysis than previously proposed ones.

5.2.5 Definition of Difficulty

The terms 'easy' and 'hard' refer to an intuitive and imprecise notion of difficulty. Unfortunately, since these terms are relative, we cannot precisely delineate this boundary, and any practical definition must be completely arbitrary.

From the perspective of the resolution proof system, the 'ideal' definition of difficulty refers to the size of the resolution refutation of an unsatisfiable formula. Unfortunately, it is meaningless for satisfiable formulas because the resolution proof system produces proofs by deriving contradictions and ultimately deriving UNSAT ($\perp$). Moreover, with finite computation resources, determining whether a polynomially-sized proof exists for a given propositional formula is impractical. Even if a polynomially-sized proof exists, we cannot guarantee that we will be able to find it.

In light of these concerns, we must consider a more practical definition. We define the difficulty of a CNF instance by the amount of time taken for a SAT solver to solve it. This definition works for both SAT and UNSAT instances, but introduces some new practical considerations.

Since we only have access to finite computing resources, we must limit the amount of time allocated to the solver for attempting to solve each instance. This will be the maximum time allowed for each instance, and serves as the boundary for classifying instances into the 'hard' end of the difficulty spectrum, where any instances which can be solved in less time are considered 'easy'. This decision is arbitrary and might not be reflective of the instances which are considered to be difficult in the future. In light of this, we choose a value of 5000 seconds[4] as a reasonable time limit.

5.3 Hierarchical Community Structure

Given that many human-developed systems are modular and hierarchical [106], it is natural to hypothesize that these properties are transferred over to Boolean formulas that capture the behaviour of such systems. We additionally hypothesize that purely randomly-generated or crafted formulas do not have these properties of hierarchy and modularity, and that this difference partly explains why solvers are efficient for the former and not for the latter class of instances. We formalize this intuition via a graph-theoretic concept called Hierarchical Community Structure (HCS), where communities can be recursively decomposed into smaller sub-communities. Although the notion of HCS has been widely studied [33, 97], it has not been considered in the context of Boolean formulas before.

Hierarchical Community Structure Definition: A *hierarchical decomposition* of a graph G is a recursive partitioning of G into subgraphs, represented as a tree T. Each node v in the tree T is labelled with a subgraph of G, with the root labelled with G itself. The children of a node corresponding to a (sub)graph H are labelled with a partitioning of H into subgraphs $\{H_1, \ldots, H_k\}$; see Figure 5.1. There are many ways to build such hierarchical decompositions. The method that we choose constructs the tree by recursively maximizing the modularity, as in the hierarchical multiresolution method [51]. We call this the HCS decomposition of a graph G: for a node v in the tree T corresponding to a subgraph H of G, we construct $|\mathcal{P}(H)|$ children, one for each of the subgraphs induced by the modularity-maximizing partition $\mathcal{P}(H)$, unless $|\mathcal{P}(H)| = 1$, in which case v becomes a *leaf* of the tree. In the case of HCS decompositions, we refer to the subgraphs labelling the nodes in the tree as *communities* of G.

We are interested in comparing the hierarchical community structures of Boolean formulas in conjunctive normal form, represented by their VIGs. For this comparison, we use

[4]This value is the time limit used by the SAT competition

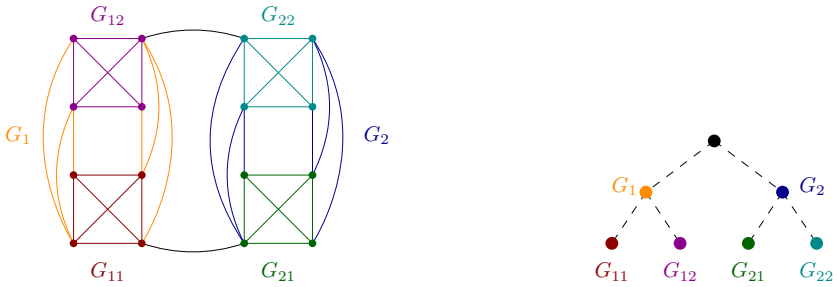

Figure 5.1: A hierarchical decomposition (right) constructed by recursively maximizing the modularity of the graph (left).

the following parameters:

- The *community degree* of a community in a HCS decomposition is the number of children of its corresponding node.
- A *leaf-community* is one with degree 0.
- The *size* of a community is its number of vertices.
- The *depth* or *level* of a community is its distance from the root.
- The *inter-community edges* of a partition $\mathcal{P}(H)$ are $E_{IC}(H) = \bigcup_{H_i,H_j \in \mathcal{P}(H)} E(H_i, H_j)$, the edges between all pairs of subgraphs, and their endpoints $V_{IC}(H) = \bigcup E_{IC}$ are the *inter-community vertices*. Note that $2|E_{IC}(H)|/|H|$ is an upper bound for the edge expansion of H.

Note that these parameters are not independent. For example, changes in the number of inter-community vertices or inter-community edges will affect modularity. Since our hierarchical decomposition is constructed using modularity, this could affect the entire decomposition and hence the other parameters.

5.4 Empirical Results

We now turn to the results of our empirical investigations with HCS parameters. We computed 49 unique parameters capturing the HCS structure, together with several base

parameters measuring different structural properties of input VIGs[5]. To compute the hierarchical community structure, we used the Louvain method [23] to detect communities and recursively call the Louvain method to produce a hierarchical decomposition. The Louvain method is considered to be more efficient and produces higher-modularity partitions than other known algorithms.

Experimental Design. In our experiments we used a set of 10 869 instances from five classes, which we believe is sufficiently large and diverse to draw sound empirical conclusions (See Appendix [68]). We did not explicitly balance the ratio of satisfiable instances in our benchmark selection because we expect our methods to be sufficiently robust as long as the benchmark contains a sufficient number of SAT and UNSAT instances.

In order to get interesting instances for modern solvers, we considered formulas which were previously used in the SAT competition from 2016 to 2018 [102]. Specifically, we took instances from five major tracks of the competition: agile, verification, crypto, crafted, and random. We also generated additional instances for some classes: for verification, we scaled the number of unrolls when encoding finite state machines for bounded model checking; for crypto, we encoded SHA-1 and SHA-256 preimage problems; for crafted, we generated combinatorial problems using `cnfgen` [67]; and for random, we generated k-CNFs at the corresponding threshold CVRs for $k \in \{3, 5\}$, again using `cnfgen`. A summary of the instances is presented in the Appendix.

We preprocessed all formulas using the MiniSAT preprocessor [41], and used MapleSAT [73] as our CDCL solver of choice since it is a leading and representative solver. The core of the preprocessing was a combination of variable elimination with subsumption and self-subsuming resolution [41]. For computing satisfiability and running time, we used SHARCNET's Intel E5-2683 v4 (Broadwell) 2.1 GHz processors [104], limiting the computation time to 5 000 seconds[6]. For parameter computation we did not limit the type of processor because structural parameter values are independent of processing power.

5.4.1 HCS-based Category Classification of Boolean Formulas

The question whether our set of HCS parameters is able to capture the underlying structure that differentiates industrial instances from the rest naturally lends itself to a classification problem. Therefore, we built a multi-class Random Forest classifier to classify a given SAT instance into one of the five categories: verification, agile, random, crafted, or crypto.

[5]For a complete list, see: https://satsolvercomplexity.github.io/hcs/data

[6]This value is the time limit used by the SAT competition.

	Category	Runtime
Score	0.996 ± 0.001	0.825 ± 0.016
Top 5 features	rootMergeability	rootInterEdges
	maxInterEdges/CommunitySize	lvl2Mergeability
	cvr	cvr
	leafCommunitySize	leafCommunitySize
	lvl2InterEdges/lvl2InterVars	lvl3Modularity

Table 5.1: Results for classification and regression experiments with HCS parameters. For regression we report R^2 values, whereas for classification we report the mean of the balanced accuracy score over 5 cross-validation datasets.

Random Forests [26] can learn complex, highly non-linear relationships while having simple structure, and hence are easier to interpret than other models (e.g., deep neural networks).

We used an off-the-shelf implementation of a Random Forest classifier implemented as `sklearn.ensemble.RandomForestClassifier` in scikit-learn [91]. Using the default set of parameters in scikit-learn version 0.24, we trained our classifier using 800 randomly sampled instances of each category on a set of 49 features to predict the class of the problem instance. We found that our classifier performs extremely well, giving an average accuracy score of 0.99 over 5 cross-validation datasets. Further, the accuracy did not depend on our choice of classifier. In particular, we found similar accuracy scores when we used C-Support Vector classification [95] instead of Random Forests.

We also determined the five most important features used by our classifier. Since several features in our feature set are highly correlated, we first performed a hierarchical clustering on the feature set based on Spearman rank-order correlations. From the 22 clusters that were generated, we arbitrarily chose a single feature from each cluster as a representative member of the cluster f[7]. Using these 22 representative features, we then computed their importance using permutation importance [26]. In Table 5.1 we list the top five representative features from each cluster, not necessarily in order of importance.

[7]See https://satsolvercomplexity.github.io/hcs/data for details on clusters.

5.4.2 HCS-based Empirical Hardness Model

We used our HCS parameters to build an empirical hardness model (EHM) to predict the run time of MapleSAT on a given instance. Since the solving time is a continuous variable, we considered a regression model built using Random Forests, namely `sklearn.ensemble.RandomForestRegressor` from scikit-learn [91]. Before training our regression model, we removed instances which timed-out at 5 000 seconds and those instances that were solved almost immediately (in zero seconds) to avoid issues with artificial cut-off boundaries. We then trained our Random Forest model using the default set of parameters in scikit-learn version 0.24 to predict the logarithm of the solving time using the remaining 1 880 instances, equally distributed between different categories.

We observed that our regression model performs quite well, with an R^2 score [107] of 0.83, which implies that in the training set, almost 83% of the variability of the dependent variable (i.e., in our case, the logarithm of the solving time) is accounted for, and the remaining 17% is still unaccounted for by our choice of parameters. Similar to category classification, we also looked for the top five predictive features used by our Random Forest regression model using the exact same process. We list the representative features in Table 5.1.

Additionally, we trained our EHM on each category of instances separately. We found that the performance of our EHM varies with instance category. Concretely, agile outperformed all other categories with an average R^2 value of 0.94, followed by random, crafted and verification instances with scores of $0.81, 0.85$ and 0.74 respectively. The worst performance was shown by the instances in crypto, with a score of 0.48.

5.4.3 HCS Parameter Value Ranges for Industrial/Random Instances

In the previous section, we reported on the top five parameters most predictive of the solver runtime in the context of our Random Forest regression model. These parameters can be divided into five distinct classes of parameters: mergeability-based, modularity-based, inter-community edge based, CVR, and leaf-community size. The parameters CVR, mergeability and modularity have been studied by previous work. CVR [30] is perhaps the most studied parameter among the three. Zulkoski et al. [121] showed that mergeability, along with combinations of other parameters, correlates well with solver run time; Ansotegui et al. [6] showed that industrial instances have good modularity compared to random instances; and Newsham et al. [89] showed that modularity has good-to-strong correlation with solver run

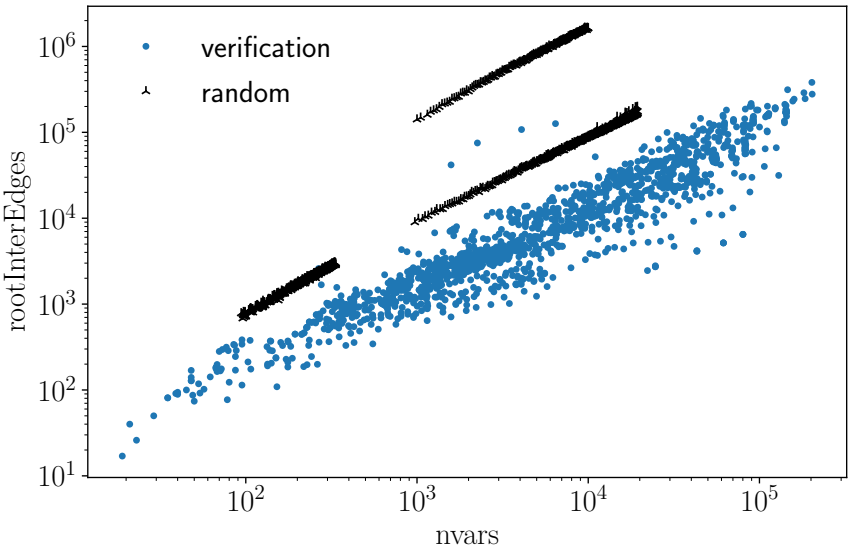

Figure 5.2: Dependence of the number of inter-community edges at the root level (rootInterEdges) vs. the number of variables in a formula, for verification and random instances in our dataset. The two distinct lines (starting from the bottom) for random instances correspond to 3-CNFs and 5-CNFs, respectively.

time. We examined the remaining parameters, i.e. inter-community edge based parameters (`rootInterEdges`) and leaf-community size to gain a better understanding of the impact of these parameters on the problem structure and solver runtime, respectively. In this subsection, we look at how HCS parameters scale as the size of industrial instances increases. And in Section 5.4.4, we introduce a HCS instance generator, which we use to perform a set of controlled experiments. We then discuss how the hardness of the instances changes when certain HCS parameters are increased/decreased.

Observations. We observe that hierarchical decomposition generally produces leaf communities of maximal size comparable to the largest clause width, except for very unbalanced formulas (easy for other reasons). The community degree is highest at root level of every instance, and seems to be bounded by $O(\log n)$. This fits within the range of parameters considered in Section 5.5.

In Figure 5.2, we show how the inter-community edge based parameter `rootInterEdges` scales with the number of variables in a formula, for verification and random instances.

We note that for random instances, `rootInterEdges` grows linearly with the instance size, whereas in verification instances it grows sublinearly. This supports our intuition that graphs of hard (random) instances are expanders, whereas graphs of industrial instances are not.

5.4.4 Scaling Experiments with HCS parameters

Instance Generator. To isolate the effects of HCS parameters on solver runtime, we built an HCS instance generator to construct SAT instances with varying leaf-community size and other HCS parameters. On a high level, the instance generator constructs instances bottom-up, starting with random disjoint formulas of predefined CVR as leaf communities, then combining them recursively by introducing bridge clauses with variables in at least two sub-communities to form super-communities at that level, which in turn are combined at the following level. We point out that in our generator, modularity is specified implicitly through the above parameters, and we do not control for mergeability at all. We refer the reader to the works by Zulkoski et al. [121] and Giráldez-Cru [45] for literature on the empirical behaviours of mergeability and power law, respectively.

It is important to note that our HCS instance generator is not intended to be perfectly representative of real-world instances. In fact, there are multiple properties of our generated instances which are not reflective of industrial instances. For example, our generator assumes that all leaf-communities have the same size and depth, which is demonstrably untrue of industrial instances. In some cases, the communities produced by our generator might not be the same as the communities which would be detected using the Louvain method to perform a hierarchical community decomposition. For example, it might be possible to further decompose the generated "leaf-communities" into smaller communities. Thus, our generator is only intended to demonstrate the effect of varying HCS parameters on solver runtime.

Observations. We constructed formulas with varying CVR, power law parameter, hierarchical degree, depth, inter-community edge density, inter-community variable density, and clause width. We found evidence which suggests that increasing any of leaf-community size, depth, or community degree, while keeping every other HCS parameter fixed, increases the overall hardness of the generated formula. For example, we found that changing the size of leaf-communities from 15 variables to 20, the solving time changed from 4.96 seconds to upwards of 5000 seconds. Similarly, changing the depth from 4 to 5 resulted in an increase in solving time from 0.03 seconds to over 5000 seconds.

5.4.5 Discussion of Empirical Results

The goal of our experimental work was to first ascertain whether HCS parameters can distinguish between industrial and random/crafted instances, and whether these parameters show any correlation with CDCL solver runtime. The robustness of our classifier indicates that HCS parameters are indeed representative of the underlying structure of Boolean formulas from different categories. Further, our empirical hardness model confirms that the correlation of HCS parameters with solver run time is strong—much stronger than previously proposed parameters. We also find that our HCS parameters are more effective in capturing the hardness or easiness of formulas from industrial/agile/random/crafted, but not crypto. The crypto class is an outlier. It is not clear from our experiments (nor any previous ones) as to why crypto instances are hard for CDCL solvers.

We also identified the top five (representative) parameters in terms of their importance in predicting the category (classification) or runtime of an instance (regression). The accuracy for classification and regression with only the top features features dropped to 0.94 and 0.77, respectively, suggesting that only a few parameters are likely to play a role in closing the question on why solvers are efficient for industrial instances. Note that a classification accuracy of 0.99 is likely to suggest that our model is over-fitting. Fortunately, in our case our models are trained over a large set of instances obtained via very different methods (e.g., random over various widths, different kinds of crafted, verification instances from different domains), and therefore, there is sufficient entropy in our data set so that overfitting is unlikely to be a concern for the robustness of our model.

In our investigation of parameters based on inter-community edges and leaf-community size, we found that industrial instances typically have small average leaf-community size, high modularity, and relatively few inter-community edges, while random/crafted have larger average leaf-community size, low modularity, and a very high number of inter-community edges. This suggests that leaf-community size and the fraction of inter-community edges, as well as community degree, are important HCS parameters to consider further.

5.5 Theoretical Results

In this section, we show that hierarchical decomposition avoids some of the pitfalls of flat community structure, a promising correlative parameter for explaining easiness of the industrial instances [89]. Community structure was theoretically shown to be insufficient by Mull et al. [84], where they showed that formulas with good community structure can have random formulas embedded in them either in a community or over the inter-community

edges. To avoid embedding a random formula in a community, its size has to be small (relative to the entire graph), and avoiding expanders over inter-community edges requires that there not be too many communities. A way to be able to restrict both is to consider a hierarchical decomposition, limiting both the number of sub-communities (community degree) in each level of the decomposition, as well as the leaf community size thus avoiding the most important issues that flat community structure suffers from.

Based on our experimental work, we narrow down the most predictive HCS parameters to be leaf-community size, community degree, and the number inter-community edges in each decomposition. These parameters also play a role in our theoretical results below. For a formula to have "good" HCS, we restrict the parameter ranges as follows: the graph must exhibit $O(\log n)$ leaf-community size and community degree, and have a small number of inter-community edges in each decomposition of a community. These assumptions are supported by our experimental results (See Appendix [68]). We show that these restrictions are necessary in Appendix, where we also present a significantly simplified proof of the result of Mull et al. [84].

Bounding the Size of Expanders in Good HCS Graphs. Ideally, we would like to be able to prove an upper bound on proof size or search time which depends on the HCS parameters of a formula. Unfortunately, our current state of understanding does not allow for that. A step towards such a result would be to show that formulas with good HCS (and associated parameter value ranges) are not susceptible to typical methods of proving resolution lower bounds. Currently, all resolution bounds exploit *expansion* properties – typically *boundary expansion* – of the CNF formula (or more precisely its bipartite constraint-variable incidence graph (CVIG)). Therefore our goal is to show that formulas with good HCS parameters have poor expansion properties, and also do not have large expanding subgraphs embedded within them. Note that the VIG is related to the CVIG by taking the square of its adjacency matrix, from where it follows that, for formulas with low width, if the VIG is not edge-expanding then the CVIG is not vertex-expanding. Furthermore, again for formulas with low width, vertex expansion is closely related to boundary expansion. Hence we only need to focus on VIG edge expansion. With this in mind, we state several positive and negative results.

First, we observe (see Appendix) that if the number of inter-community edges at the top level of the decomposition grows sub-linearly with n and at least two sub-communities contain a constant fraction of vertices, then this graph family is not an expander. Unfortunately, we can also show (see Appendix) that graphs with good HCS can simultaneously have sub-graphs that are large expanders, with the worst case being very sparse expanders,

capable of "hiding" in the hierarchical decomposition by contributing relatively few edges to any cut. To avoid that, we require an explicit bound on the number of inter-community edges, in addition to small community degree and small leaf-community size. This lets us prove the following statement.

Theorem 9. *Let $G = \{G_n\}$ be a family of graphs. Let $f(n) \in \omega(poly(\log n))$, $f(n) \in O(n)$. Assume that G has HCS with the number of inter-community edges $o(f(n))$ for every community C of size at least $\Omega(f(n))$ and depth is bounded by $O(\log n)$. Then G does not contain an expander of size $f(n)$ as a subgraph.*

Note that our experiments show that the leaf size and depth in industrial instances are relatively small and the number of inter-community edges grows slowly. From this and the theorem above, we can show that graphs with *very* good HCS properties do not contain linear-sized expanders.

Lower Bounds Against HCS: We are also able to show several of strong lower bounds on formulas with good HCS (see Appendix). For a number of combinations of parameters, we show that restricting ourselves to "good" ranges of these parameters does not rule out formulas which require superpolynomial size resolution refutations. Our most striking counterexample essentially shows that if the degree of the VIG is more than a small constant, then it is possible to embed formulas of superpolynomial resolution complexity. In contrast with the previous results on the size of embeddable expanders in instances with good HCS, this result shows how to embed a sparse expander of superlogarithmic size.

Hierarchical vs. Flat Modularity: It is well-known that modularity suffers from a *resolution limit* and cannot detect communities smaller than a certain threshold [43], and that HCS can avoid this problem in some instances [23]. In Appendix we provide an asymptotic, rigorous statement of this observation.

Theorem 10. *There exists a graph G whose natural communities are of size $\log(n)$ and correspond to the (leaf) HCS communities, while the partition maximizing modularity consists of communities of size $\Theta\left(\sqrt{n/\log^3 n}\right)$.*

5.6 Related Work

Community Structure: Using modularity to measure community structure allows one to distinguish industrial instances from randomly-generated ones [6]. Unfortunately, it has

been shown that expanders can be embedded within formulas with high modularity [84], i.e., there exist formulas that have good community structure and yet are hard for solvers.

Heterogeneity: Unlike uniformly-random formulas, the variable degrees in industrial formulas follow a powerlaw distribution [5]. However, degree heterogeneity alone fails to explain the hardness of SAT instances. Some heterogeneous random k-SAT instances were shown to have superpolynomial resolution size [22], making them intractable for current solvers.

SATzilla: SATzilla uses 138 disparate parameters [118], some of which are probes aimed at capturing a SAT solver's state at runtime, to predict solver running time. Unfortunately, there is little or no evidence that most of these parameters are amenable to theoretical analysis.

Clause-Variable Ratio (CVR): Cheeseman et al. [30] observed the satisfiability threshold behavior for random k-SAT formulas, where they show formulas are harder when their CVR are closer to the satisfiability threshold. Outside of extreme cases, CVR alone seems to be insufficient to explain hardness (or easiness) of instances, as it is possible to generate both easy and hard formulas with the same CVR [44]. Satisfiability thresholds are poorly defined for industrial instances, and Coarfa et al. [34] demonstrated the existence of instances for which the satisfiability threshold is not equal to the hardness threshold.

Treewidth: Although there are polynomial-time non-CDCL algorithms for SAT instances with bounded treewidth [3], treewidth by itself does not appear to be a predictive parameter of CDCL solver runtime. For example, Mateescu [81] showed that some easy instances have large treewidth, and later it was shown that treewidth alone does not seem to correlate well with solving time [121].

Backdoors: In theory, the existence of small backdoors [116, 100] should allow CDCL solvers to solve instances quickly, but empirically backdoors have been shown not to strongly correlate with CDCL solver run time [62].

5.7 Conclusions and Future Work

In this chapter, we propose HCS as a correlative set of parameters for explaining the power of CDCL SAT solvers over industrial instances, which also has good theoretical properties. Empirically, HCS parameters are much more predictive than previously proposed correlative parameters in terms of classifying instances into random/crafted vs. industrial, and in terms of predicting solver run time. Among the top five most predictive parameters,

three are HCS parameters, namely leaf-community size, modularity and fraction of inter-community edges. The remaining two are cvr and mergeability. We further identify the following core HCS parameters that are the most predictive among all HCS parameters, namely, leaf-community size, modularity, and fraction of inter-community edges. Indeed, these same parameters also play a role in our subsequent theoretical analysis, where we show that counterexamples to flat community structure do not apply to HCS, and that restricting certain HCS parameters limits the size of embeddable expanders. In the final analysis, we believe that HCS, along with other parameters such as mergeability or heterogeneity, will play a role in finally settling the question of why solvers are efficient over industrial instances.

Chapter 6

Adaptive Invocation of SDCL through Reinforcement Learning: A VSIDS-inspired Approach

In our preceding paper, we introduced MapleSDCL, a pioneering Satisfaction-Driven Clause Learning (SDCL) SAT solver. A rigorous empirical evaluation attested to its proficiency, particularly in swiftly solving Mutilated Chess Board (MCB) problems, outpacing traditional CDCL solvers without necessitating alterations to the branching heuristic of the foundational CDCL SAT solver. While MapleSDCL demonstrated commendable performance, our insights revealed an opportunity: optimizing the accuracy of SDCL invocations.

This chapter pivots around the implementation and evaluation of a reinforcement learning (RL) framework tailored to enhance the efficiency of SDCL invocations. Given the associated computational cost of invoking SDCL and the observation that every invocation does not invariably result in clause learning, we propose an RL-based model to ascertain scenarios where SDCL invocation is most promising.

6.1 Introduction

Conflict-Driven Clause Learning (CDCL) SAT solvers are routinely used to solve large industrial problems obtained from a variety of applications in software engineering [29], formal methods [32], security [40, 117] and, AI [24], even though the underlying Boolean

satisfiability (SAT) problem is well known to be NP-complete [35] and believed to be intractable in general. Despite this, solver research has made significant progress in improving CDCL solvers' components and heuristics [79].

It is well known that CDCL SAT solvers are polynomially equivalent to resolution [94, 8], and consequently, it follows that classes of formulas, such as the pigeon hole principle (PHP), that are hard for resolution are also hard for CDCL SAT solvers. To address such limitations, researchers are actively designing and implementing solvers that correspond to stronger propositional proof systems.

One such class of solvers is called Satisfaction-Driven Clause Learning (SDCL) solvers [57, 56, 55], which are based on the propagation redundancy (PR) property [54, 56]. The SDCL paradigm extends CDCL in the following way: unlike CDCL solvers, SDCL solvers may learn clauses even when an assignment trail α is consistent. To be more precise, an SDCL solver first computes a new formula $P_\alpha(F)$, known as a pruning predicate. Then, it checks the satisfiability of $P_\alpha(F)$. If it is satisfiable, it means $\neg\alpha$ is *redundant* with respect to the formula, and the solver can learn the clause $(\neg\alpha)$. Even though the intuition is clear and procedures for computing a possible $P_\alpha(F)$ are very well defined, it is still an extremely challenging task to automate SDCL.

There are two main problems in this setting: first, the satisfiability check for the formula $P_\alpha(F)$ is NP-complete and is hard to solve in general. It essentially requires the SDCL solver to call another SAT solver that we refer to as a sub-solver. Given that this sub-solver call can be expensive, one needs to be strategic about when to invoke it during the run of an SDCL solver. Second, the clauses learned by SDCL can be large, and we want to learn shorter clauses whenever possible.

While the latter has been tackled in our prior research [90], this chapter zeroes in on the former problem, aiming to heighten the success rate of SDCL invocations. We introduce an approach inspired by the VSIDS branching heuristic. In essence, each variable is assigned an activity score reflecting its recent usefulness. Variables pivotal in recent CDCL learnt clauses receive higher activity scores, guiding the solver to prioritize them in subsequent attempts. For SDCL, we maintain a distinct set of scores, and we award variables that lead to successful SDCL invocations.

6.2 Propagation Redundancy and SDCL

Despite their success in a variety of real-world applications [105, 96, 21, 58], CDCL SAT solvers have well-known limitations. Proof complexity techniques have established the

polynomial equivalence between CDCL and general resolution [94, 8], the proof system with the inference rule that allows one to derive $C \vee D$ given two clauses of the form $l \vee C$ and $\neg l \vee D$. An important consequence of this equivalence is that if an unsatisfiable formula does not have a polynomial size proof by resolution, no run of CDCL can determine the unsatisfiability of the formula in polynomial time.

6.2.1 Propagation Redundancy

This limitation has motivated the search for extensions of CDCL solvers that may allow the resultant method to simulate more powerful proof systems. One example is the extended resolution proof system [111]: by allowing the introduction of new variables to resolution, it can produce polynomial size proofs of the pigeon-hole principle [36], which requires exponential-size resolution proofs otherwise. However, adding new variables would exponentially increase the search space of the formula. A newer direction [54, 56] tries to avoid the addition of new variables, and is instead based on the well-known notion of redundancy:

Definition 10. *A clause C is* **redundant** *with respect to a formula F if F and $F \wedge C$ are equisatisfiable.*

To provide a more useful characterization of redundancy, we need some definitions.

Definition 11. *Given an assignment α and a clause C, we define $\mathbf{C}|_\alpha = \top$ if $\alpha \models C$; otherwise $\mathbf{C}|_\alpha$ is the clause consisting of all literals of C that are undefined in α. For a formula F, we define the formula $\mathbf{F}|_\alpha = \{C|_\alpha \mid C \in F \text{ and } \alpha \not\models C\}$.*

Theorem 11 ([54], Theorem 1)**.** *A non-empty clause C is redundant with respect to a formula F if and only if there exists an assignment ω such that $\omega \models C$ and $F \wedge \neg C \models F|_\omega$.*

From a practical point of view, this characterization does not help much, because even if we know ω (known as the *witness*) it is hard to check whether the property holds. This is why a more limited notion of redundancy has been defined [54]:

Definition 12. *A clause C is* **propagation redundant (PR)** *with respect to a formula F if there exists an assignment ω such that $\omega \models C$ and $F \wedge \neg C \vdash_1 F|_\omega$*

Note that since $F \wedge \neg C \vdash_1 F|_\omega$ implies $F \wedge \neg C \models F|_\omega$, any PR clause is redundant. Hence, we can add PR clauses to our formula to make it easier to solve without affecting its

satisfiability. If we force ω to assign all variables in C but no other variable, we can obtain weaker but simpler notions of redundancy: if we force ω to satisfy exactly one literal of C, we obtain *literal-propagation redundant (LPR)* clauses; if allow ω to satisfy more than one literal of C, we obtain *set-propagation redundant (SPR)* clauses. Obviously, any LPR clause is SPR, and any SPR clause is PR, but none of these three notions are equivalent as the following examples show.

Example 1 ([54]). *Let* $F = \{x \vee y,\ \ x \vee \neg y \vee z,\ \ \neg x \vee z,\ \ \neg x \vee u,\ \ x \vee \neg u\}$ *and* $C = x \vee u$. *The witness* $\omega = \{x, u\}$ *satisfies* C *and, since* $F|_\omega = \{z\}$, *it holds that* $F \wedge \neg C \vdash_1 F|_\omega$, *that is, unit propagation on* $F \wedge \neg x \wedge \neg u \wedge \neg z$ *results in a conflict. Hence,* C *is SPR w.r.t.* F.

However, it is not LPR. The reason is that there are only two possible witnesses that satisfy exactly one literal of C: $\omega_1 = \{x, \neg u\}$ *and* $\omega_2 = \{\neg x, u\}$. *But we have both* $F|_{\omega_1}$ *and* $F|_{\omega_2}$ *contain, among others, the empty clause. Hence,* $F \wedge \neg C \vdash_1 F|_{\omega_1}$ *and* $F \wedge \neg C \vdash_1 F|_{\omega_2}$ *require that unit propagation on* $F \wedge \neg C$, *that is,* $F \wedge \neg x \wedge \neg u$, *results in a conflict, which is not the case.*

Example 2 ([54]). *Let* $F = \{x \vee y, \neg x \vee y, \neg x \vee z\}$ *and* $C = (x)$. *If we consider the witness* $\omega = \{x, z\}$, *we have that* $F|_\omega = \{y\}$. *It is obvious that* $\omega \models C$ *and also* $F \wedge \neg x \vdash_1 y$. *Thus,* C *is PR w.r.t.* F. *However it is not SPR because the only possible witness would be* $\omega_1 = \{x\}$, *but* $F|_{\omega_1} = \{y, z\}$ *and it does not hold that* $F \wedge \neg x \vdash_1 z$.

6.2.2 SDCL and Reducts

It was proved in [54] that the proof system that combines resolution with the addition of PR clauses admits polynomial-sized proofs for the pigeon hole principle. However, it is not a trivial task to add this capability to CDCL solvers. This question was addressed with the development of Satisfiability-Driven Clause Learning (SDCL) [57]. The key notion in this new solving paradigm is the one of *pruning predicate*:

Definition 13. *Let* F *be a formula and* α *an assignment. A* **pruning predicate** *for* F *and* α *is a formula* $P_\alpha(F)$ *such that if it is satisfiable, then the clause* $\neg\alpha$ *is redundant w.r.t.* F.

SDCL extends CDCL in the following way (See also Algorithm 6.2.2). Before making a decision, a pruning predicate for the assignment α and formula F is constructed. If satisfiable, we can learn $\neg\alpha$ and use it for backjump and continuing the search, hence pruning away the search tree without needing to find a conflict. This leads to the simple code in Algorithm 6.2.2, where removing lines 9 to 12 results in the standard CDCL

Algorithm 2 The SDCL algorithm. Note that removing lines 11-14 results in the CDCL algorithm.

```
1:  α := ∅
2:  while true do
3:    α := unitPropagate(F, α)
4:    if conflict found then
5:      C := analyzeConflict()
6:      F := F ∧ C
7:      if C is the empty clause then
8:        return UNSAT
9:      end if
10:     α := backjump(C, α)
11:   else if P_α(F) is satisfiable then
12:     C := analyzeWitness()
13:     F := F ∧ C
14:     α := backjump(C, α)
15:   else
16:     if all variables are assigned then
17:       return SAT
18:     end if
19:     α := α ∪ Decide()
20:   end if
21: end while
```

algorithm, and where we can assume, for simplicity, that $analyzeWitness()$ returns $\neg\alpha$. More sophisticated versions of $analyzeWitness$ are discussed in the next Section.

We can understand SDCL as a parameterized algorithm, since the use of different pruning predicates $P_\alpha(F)$ leads to distinct types of SDCL algorithms with possibly different underlying proof systems. In the following, we summarize the contributions of [57, 55] and explain the different pruning predicates and the corresponding proof systems that are known.

Definition 14. *Given formula F and a (partial) assignment α, the* **positive reduct** *$p_\alpha(F)$ is the formula $\neg\alpha \wedge G$, where $G = \{touched_\alpha(D) \mid D \in F \text{ and } \alpha \models D\}$.*

That is, we only consider clauses satisfied by α, and among them, only the literals that are assigned. In [57] it is proved that $p_\alpha(F)$ is a valid pruning predicate. Moreover, a

precise characterization of the redundancy achieved by $p_\alpha(F)$ is given: $p_\alpha(F)$ is satisfiable if and only if $\neg\alpha$ is set-blocked in F.

Definition 15. *A clause C is* **set-blocked** *in a formula F if there exists a subset $L \subseteq C$ such that, for every clause D containing the negation of some literal in C, the clause $(C \setminus L) \vee \neg L \vee D$ contains two complementary literals.*

The results in [57] imply that a proof system based on resolution and set-blocked clauses has polynomial size proofs for the pigeon hole principle. It is also known [54] that set-blocked clauses are a particular case of SPR clauses. If one wants to obtain the full power of SPR clauses, the following pruning predicate is needed:

Definition 16. *Given formula F and a (partial) assignment α, the* **filtered positive reduct** *$f_\alpha(F)$ is the formula $\neg\alpha \wedge G$, where $G = \{touched_\alpha(D) \mid D \in F \text{ and } F \wedge \alpha \not\vdash_1 untouched_\alpha(D)\}$.*

Again, a precise characterization of the power of $f_\alpha(F)$ is known [55]: $f_\alpha(F)$ is satisfiable if and only if $\neg\alpha$ is SPR with respect to F. Despite being harder to compute than $p_\alpha(F)$, the fact that $f_\alpha(F)$ is a subset of the clauses in $p_\alpha(F)$ makes it easier to check for satisfiability. Finally, another pruning predicate is given in [55] that achieves the full power of PR clauses, but it is not considered to be practical. We close this sequence of pruning predicates and their corresponding redundancy characterization with a novel pruning predicate and its corresponding redundancy notion.

Definition 17. *Given formula F and a (partial) assignment α, the* **purely positive reduct** *$pp_\alpha(F)$ is the formula $\neg\alpha \wedge G$, where $G = \{satisfied_\alpha(D) \mid D \in F \text{ and } \alpha \models D\}$.*

Since all clauses in $pp_\alpha(F)$ are subclauses of clauses in $p_\alpha(F)$, whenever $pp_\alpha(F)$ is satisfiable, $p_\alpha(F)$ is also satisfiable. This proves that $pp_\alpha(F)$ is a pruning predicate, but we can be more precise about the notion of redundancy it corresponds to.

Definition 18. *We say that a literal $l \in C$* **blocks** *C in F if an only if for every clause D in F containing literal $\neg l$, resolution between C and D gives a tautology. A clause C is* **blocked** *in F if and only if there exists some literal $l \in C$ that blocks C in F.*

Theorem 12. *Given a formula F and an assignment α, the formula $pp_\alpha(F)$ is satisfiable if and only if the clause $\neg\alpha$ is blocked in F.*

Proof. Left to right: let β be a model of $pp_\alpha(F)$. Since $\beta \models \neg\alpha$, we can take any literal $\neg l$ in $\neg\alpha$ satisfied by β. We now prove that $\neg l$ blocks $\neg\alpha$ in F. Let us consider a clause

of the form $l \vee C \in F$. Since $l \in \alpha$ we have that $\alpha \models l \vee C$, and hence there is a clause of the form $l \vee satisfied_\alpha(C)$ in $pp_\alpha(F)$. Since $\beta \models pp_\alpha(F)$ and $\beta \models \neg l$, necessarily $\beta \models satisfied_\alpha(C)$. This means that C contains a literal from α different from l, and hence if we apply resolution between the clause $\neg\alpha$ and $l \vee C$ we obtain a tautology.

Right to left: Assume w.l.o.g. that the clause $\neg\alpha$ is blocked w.r.t. $\neg l$ in F. We prove that $\hat{\alpha} := \alpha \setminus \{l\} \cup \{\neg l\}$ is a model of $pp_\alpha(F)$. It is obvious that $\hat{\alpha}$ satisfies the clause $\neg\alpha \in pp_\alpha(F)$. Any other clause $D \in pp_\alpha(F)$ is of the form $satisfied_\alpha(C)$ for some $C \in F$ such that $\alpha \models C$. There are now in principle two cases:

1. if D is not the unit clause l, it necessarily contains a literal from α different from l, and hence $\hat{\alpha}$ satisfies it.

2. If D is the unit clause l, this means that clause $C \in F$ does not contain any literal from α except for l. Thus, applying resolution between $\neg\alpha$ and C cannot give a tautology, contradicting the fact that $\neg\alpha$ is blocked w.r.t $\neg l$ in F. Hence, this case cannot take place.

□

We finish this section with one important remark about the computation of reducts in SDCL: we need to add all already computed redundant clauses in the reduct computation when trying to find additional ones. Let us show why not doing this is incorrect. Given the satisfiable formula $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_4)$, the SDCL solver might first build the assignment $\alpha = \{x_1, \neg x_2\}$. Its positive reduct is $(\neg x_1 \vee x_2) \wedge (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$, which is satisfiable, and hence we learn the redundant clause $\neg x_1 \vee x_2$. If the solver now builds the assignment $\{\neg x_1, x_2\}$, the positive reduct w.r.t F is $(x_1 \vee \neg x_2) \wedge (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$, which is again satisfiable and allows us to learn the clause $x_1 \vee \neg x_2$. However, adding the two learned redundant clauses to F makes it unsatisfiable. The solution is to build the second positive reduct w.r.t. F conjuncted with the first learned redundant clause. The corresponding reduct is $(x_1 \vee \neg x_2) \wedge (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2)$, which is now unsatisfiable and hence does not allow us to learn the second redundant clause.

A natural question that arises now is whether we also need to add all clauses that were derived using CDCL-style conflict analysis in a reduct. The answer is that we do not need to do so. The reason is that, given two formulas $G_1 \equiv G_2$, it holds that C is redundant w.r.t. G_1 if and only if C is redundant w.r.t G_2. Now, if the current formula that the SDCL solver has in its database is $F \wedge L \wedge R$, where F is the original formula, L are the lemmas derived by CDCL-style conflict analysis and R are the learned redundant clauses, it holds

that $F \wedge L \wedge R \equiv F \wedge R$. Therefore, it is sufficient to compute redundant clauses w.r.t. $F \wedge R$ only. Having said that, it is better to compute reduntant clauses w.r.t $F \wedge R \wedge U$, where U denotes CDCL-derived unit clauses, because it results in smaller reducts and faster sub-solver calls. Note that for correctness, clauses in R are never deleted. This design decision prevents us from using off-the-shelf proof checkers like dpr-trim[1].

6.3 Experimental Evaluation

As we try to refine Satisfaction-Driven Clause Learning (SDCL) solvers, we have drawn upon the proven strengths of the VSIDS branching heuristic, specifically from its implementation in MapleSAT. The integration was straightforward; we directly transplanted MapleSAT's VSIDS code to assign SDCL activity scores to variables. This, however, brings forth a complex challenge: translating the aggregated information from these activity scores into actionable decisions.

Central to every SDCL invocation's success is the assignment trail, a series of variable assignments that directly influence the outcome of the SDCL call. If one envisions each variable assignment as carrying a weight or "score" denoting its relevance, the cumulative weight of the trail could potentially inform the likelihood of a successful SDCL invocation. The crux of our exploration revolves around this aggregation. Our primary objective is to devise a mechanism that evaluates the combined activity scores of all variables on a given trail and translates this aggregated value into a binary decision—invoke SDCL or abstain.

There are two challenges we need to address:

- Score Aggregation: Combining individual variable scores, focusing on ensuring that the aggregated value is a reliable predictor of SDCL invocation success.
- Threshold Determination: Identifying optimal cut-off values for the aggregated score, above which SDCL invocation is deemed promising and below which it's considered prudent to avoid.

In the process of identifying an optimal trail for SDCL invocation, a comparison is made between the average activity score and the median activity score of the variables within that trail. The reasoning for this comparison is rooted in the nature of distribution:

[1] https://github.com/marijnheule/dpr-trim

- When the average score exceeds the median, the distribution of activity scores is skewed to the right. This suggests that there is a greater presence of variables with higher scores in the trail, indicating these are "beneficial" or "advantageous" variables.
- Conversely, when the average is less than the median, the distribution leans to the left, implying a deficit of such advantageous variables.

However, an empirical observation has been made that a direct comparison between the average and median can lead to excessive invocations of SDCL in certain problem instances. This frequent reliance on SDCL can introduce considerable overhead, affecting the efficiency of the solution process.

To address this, an adjustment has been proposed. Rather than comparing the average directly to the median, the average is multiplied by a factor "d" (where $0 < d < 1$) prior to the comparison. This modification aims to optimize the number of SDCL calls, ensuring a balance between solver accuracy and operational efficiency.

In Figure 6.1, we see that the overall success rate of SDCL calls goes down as the underlying MCB problems scale and get harder, regardless of the value of d. However, we also observe that $d = 0.95$ seems to improve the success rate for SDCL calls the most for MCB problems. On the other hand, we observe that for randomG instances, $d = 0.95$ becomes a poorly performing configuration in terms of SDCL success rate. This suggests that there is no fixed d value that works across all the benchmarks under consideration. Thus, motivating for a better and more adaptive way to choose a d value. In the same time, the observation above suggests that there is some merit to the idea of using VSIDS like mechanism to account for the usefulness of trail variables in the context of SDCL.

6.4 Conclusions and Future Work

The integration of the VSIDS branching heuristic into the Satisfaction-Driven Clause Learning (SDCL) framework presents a promising avenue for enhancing solver performance. By assigning activity scores to variables, akin to the VSIDS methodology, and adapting their aggregation based on a balance between average and median values, a more discerning strategy for SDCL invocation is proposed. While the direct comparison of average and median activity scores led to excessive SDCL calls in specific instances, introducing a modifying factor 'd' has shown potential in optimizing these invocations. This synergy between VSIDS and SDCL not only amplifies the precision of SDCL solver decisions but also

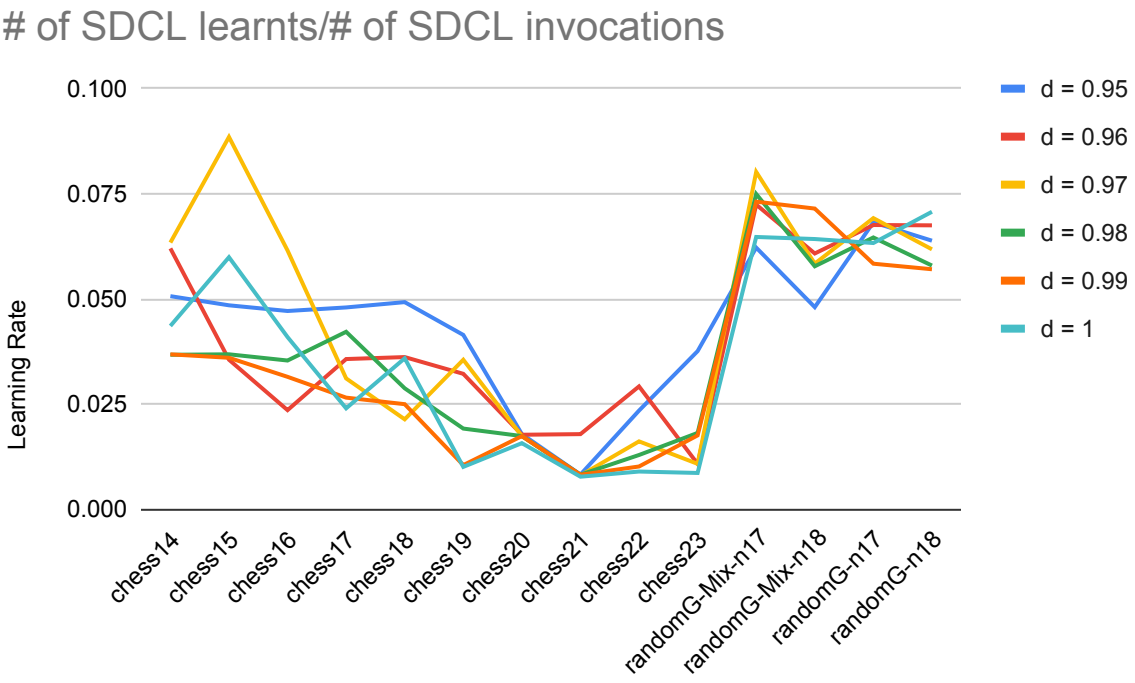

Figure 6.1: By varying the factor "d", the success of SDCL invocations changes for different instances.

underscores the value of merging established heuristics with innovative solution methods. Also, the trends observed in the experiments motivate a smart and adaptive strategy for determining the value of the 'd' factor.

Chapter 7

Conclusion

In conclusion, this thesis has made contributions to both the theoretical understanding and practical enhancement of SAT solvers. The research addresses some of the open problems in the critical challenge of bridging the gap between a theoretical understanding of the intricacies of SAT solver efficiency and their empirical success. The thesis also proposes several machine learning based techniques which improve solver performance on wide range of benchmarks, thus pushing forward the state-of-the-art in solver heuristics.

The theoretical contributions include an understanding of the power of restarts in various models of SAT solvers, providing results such as exponential separations between solver configurations with and without restarts. These findings not only contribute to the theoretical foundations of SAT solving but also lead to the development of a practical restart policy that outperforms state-of-the-art solvers on challenging instances, including those derived from bitcoin mining problems.

Building upon the theoretical insights into restarts, the thesis introduces the concept of resets and formulates the problem of choosing between restarts with and without resets as a multi-armed bandit (MAB) problem. A reinforcement learning (RL) based reset policy is proposed, demonstrating superior performance on bitcoin mining benchmarks and maintaining competitiveness against baseline solvers in SAT competition instances.

Furthermore, the research explores the hierarchical community structure (HCS) of Boolean formulas, using parameters derived from graph partitioning to develop an Empirical Hardness Model. The HCS parameters are shown to be strongly correlated with solver run time, leading to the creation of a classifier that accurately distinguishes between easy industrial and hard random/crafted instances. The scalability studies confirm our theory of HCS and provide more empirical evidence for our theory.

Finally, the thesis also presents novel techniques in the context of satisfaction-driven clause-learning (SDCL) solvers, a new class of solvers known to be exponentially stronger than CDCL solvers. Despite the theoretical power of SDCL, there are many challenges in automating and determinizing such solvers. To address this, we propose machine learning techniques to help decide when to invoke a SDCL subroutine, where their goal is to invoke the SDCL sub-solver in a strategic way to reduce the associated overhead. The resulting SDCL solver, integrated with MaxSAT techniques and conflict analysis, surpasses existing solvers on certain combinatorial benchmarks, specifically the Mutilated Chess Board (MCB) problems.

In summary, this thesis contributes by combining theoretical advancements, practical innovations, and the application of machine learning to enhance solver performance. The insights gained not only deepen our understanding of the underlying principles of SAT solving, but also offer practical solutions that push the boundaries of solver capabilities in addressing complex real-world problems.

References

[1] Michael Alekhnovich, Edward A. Hirsch, and Dmitry Itsykson. Exponential lower bounds for the running time of DPLL algorithms on satisfiable formulas. *Journal of Automated Reasoning*, 35(1-3):51–72, 2005.

[2] Michael Alekhnovich, Jan Johannsen, Toniann Pitassi, and Alasdair Urquhart. An exponential separation between regular and general resolution. *Theory of Computing*, 3(1):81–102, 2007.

[3] Michael Alekhnovich and Alexander Razborov. Satisfiability, Branch-Width and Tseitin Tautologies. *computational complexity*, 20(4):649–678, December 2011.

[4] Carlos Ansótegui, Maria Luisa Bonet, Jesús Giráldez-Cru, and Jordi Levy. The Fractal Dimension of SAT Formulas. In *Proceedings of the 7th International Joint Conference on Automated Reasoning - IJCAR 2014*, pages 107–121, 2014.

[5] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. Towards Industrial-Like Random SAT Instances. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pages 387–392, 2009.

[6] Carlos Ansótegui, Jesús Giráldez-Cru, and Jordi Levy. The Community Structure of SAT Formulas. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing - SAT 2012*, pages 410–423, 2012.

[7] Albert Atserias, Johannes Klaus Fichte, and Marc Thurley. Clause-learning algorithms with many restarts and bounded-width resolution. *Journal of Artificial Intelligence Research*, 40:353–373, 2011.

[8] Albert Atserias, Johannes Klaus Fichte, and Marc Thurley. Clause-learning algorithms with many restarts and bounded-width resolution. *J. Artif. Intell. Res.*, 40:353–373, 2011.

[9] Albert Atserias and Moritz Müller. Automating resolution is np-hard. *Journal of the ACM (JACM)*, 67(5):1–17, 2020.

[10] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Twenty-first international joint conference on artificial intelligence*. Citeseer, 2009.

[11] Gilles Audemard and Laurent Simon. Refining restarts strategies for SAT and UNSAT. In *International Conference on Principles and Practice of Constraint Programming*, pages 118–126. Springer, 2012.

[12] Florent Avellaneda. A short description of the solver EvalMaxSAT. In Fahiem Bacchus, Jeremias Berg, Matti Järvisalo, and Ruben Martins, editors, *MaxSAT Evaluation 2020*, pages 8–9, 2020.

[13] Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res.*, 22:319–351, 2004.

[14] Paul Beame and Toniann Pitassi. Propositional proof complexity: Past, present, and future. In Gheorghe Paun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Current Trends in Theoretical Computer Science, Entering the 21th Century*, pages 42–70. World Scientific, 2001.

[15] Eli Ben-Sasson and Avi Wigderson. Short proofs are narrow—resolution made simple. *Journal of the ACM*, 48(2):149–169, 2001.

[16] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

[17] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

[18] Armin Biere and Andreas Fröhlich. Evaluating CDCL variable scoring schemes. In Marijn Heule and Sean A. Weaver, editors, *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, volume 9340 of *Lecture Notes in Computer Science*, pages 405–422. Springer, 2015.

[19] Armin Biere and Andreas Fröhlich. Evaluating cdcl variable scoring schemes. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 405–422. Springer, 2015.

[20] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[21] Armin Biere and Daniel Kröning. Sat-based model checking. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 277–303. Springer, 2018.

[22] Thomas Bläsius, Tobias Friedrich, Andreas Göbel, Jordi Levy, and Ralf Rothenberger. The Impact of Heterogeneity and Geometry on the Proof Complexity of Random Satisfiability. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021*, pages 42–53, 2021.

[23] Vincent Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast Unfolding of Communities in Large Networks. *Journal of Statistical Mechanics Theory and Experiment*, 2008, April 2008.

[24] Avrim L Blum and Merrick L Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1-2):281–300, 1997.

[25] 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.

[26] Leo Breiman. Random Forests. *Mach. Learn.*, 45(1):5–32, October 2001.

[27] Samuel R. Buss, Jan Hoffmann, and Jan Johannsen. Resolution trees with lemmas: Resolution refinements that characterize DLL algorithms with clause learning. *Logical Methods in Computer Science*, 4(4), 2008.

[28] Samuel R. Buss and Leszek Kołodziejczyk. Small stone in pool. *Logical Methods in Computer Science*, 10(2):16:1–16:22, June 2014.

[29] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. EXE: Automatically Generating Inputs of Death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):1–38, 2008.

[30] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the Really Hard Problems Are. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, IJCAI'91, pages 331–337, 1991.

[31] Mohamed Sami Cherif, Djamal Habet, and Cyril Terrioux. Combining vsids and chb using restarts in sat. In *27th International Conference on Principles and Practice of Constraint Programming*, 2021.

[32] Edmund M Clarke Jr, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. *Model Checking*. MIT press, 2018.

[33] Aaron Clauset, Cristopher Moore, and M. E. J. Newman. Hierarchical Structure and the Prediction of Missing Links in Networks. *Nature*, 453(7191):98–101, May 2008.

[34] Cristian Coarfa, Demetrios D. Demopoulos, Alfonso San Miguel Aguirre, Devika Subramanian, and Moshe Y. Vardi. Random 3-SAT: The Plot Thickens. *Constraints*, 8(3):243–261, July 2003.

[35] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[36] Stephen A. Cook. A short proof of the pigeon hole principle using extended resolution. *SIGACT News*, 8(4):28–32, 1976.

[37] Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *The Journal of Symbolic Logic*, 44(1):36–50, 1979.

[38] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[39] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[40] Julian Dolby, Mandana Vaziri, and Frank Tip. Finding Bugs Efficiently With a SAT Solver. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 195–204, 2007.

[41] Niklas Eén and Armin Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing*, pages 61–75, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[42] ABKFM Fleury and Maximilian Heisinger. Cadical, kissat, paracooba, plingeling and treengeling entering the sat competition 2020. *SAT COMPETITION*, 2020:50, 2020.

[43] Santo Fortunato and Marc Barthélemy. Resolution Limit in Community Detection. *Proceedings of the National Academy of Sciences*, 104(1):36–41, 2007.

[44] Tobias Friedrich, Anton Krohmer, Ralf Rothenberger, and Andrew M. Sutton. Phase Transitions for Scale-Free SAT Formulas. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, page 3893–3899. AAAI Press, 2017.

[45] Jesús Giráldez-Cru. *Beyond the Structure of SAT Formulas*. PhD thesis, Universitat Autònoma de Barcelona, 2016.

[46] Jesús Giráldez-Cru and Jordi Levy. A Modularity-Based Random SAT Instances Generator. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015*, pages 1952–1958, 2015.

[47] Eugene Goldberg and Yakov Novikov. Berkmin: A fast and robust sat-solver. *Discrete Applied Mathematics*, 155(12):1549–1561, 2007.

[48] 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, 2000.

[49] Carla P. Gomes, Bart Selman, and Henry A. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI 1998)*, pages 431–437, 1998.

[50] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1994.

[51] Clara Granell, Sergio Gomez, and Alex Arenas. Hierarchical Multiresolution Method to Overcome the Resolution Limit in Complex Networks. *International journal of bifurcation and chaos*, 22(07):1250171, 2012.

[52] Shai Haim and Toby Walsh. Restart strategy selection using machine learning techniques. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 312–325. Springer, 2009.

[53] Philipp Hertel, Fahiem Bacchus, Toniann Pitassi, and Allen Van Gelder. Clause learning can effectively p-simulate general propositional resolution. In Dieter Fox and Carla P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, (AAAI 2008)*, pages 283–290. AAAI Press, 2008.

[54] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Short proofs without new variables. In Leonardo de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 130–147. Springer, 2017.

[55] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Encoding redundancy for satisfaction-driven clause learning. In Tomás Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I*, volume 11427 of *Lecture Notes in Computer Science*, pages 41–58. Springer, 2019.

[56] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Strong extension-free proof systems. *J. Autom. Reason.*, 64(3):533–554, 2020.

[57] Marijn J. H. Heule, Benjamin Kiesl, Martina Seidl, and Armin Biere. Pruning through satisfaction. In Ofer Strichman and Rachel Tzoref-Brill, editors, *Hardware and Software: Verification and Testing - 13th International Haifa Verification Conference, HVC 2017, Haifa, Israel, November 13-15, 2017, Proceedings*, volume 10629 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2017.

[58] 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 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 228–245. Springer, 2016.

[59] Marijn JH Heule, Benjamin Kiesl, and Armin Biere. Encoding redundancy for satisfaction-driven clause learning. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 41–58. Springer, 2019.

[60] Marijn JH Heule, Benjamin Kiesl, Martina Seidl, and Armin Biere. Pruning through satisfaction. In *Haifa Verification Conference*, pages 179–194. Springer, 2017.

[61] Shlomo Hoory, Nathan Linial, and Avi Wigderson. Expander Graphs and Their Applications. *Bulletin of the American Mathematical Society*, 43(4):439–561, 2006.

[62] Philip Kilby, John Slaney, Sylvie Thiebaux, and Toby Walsh. Backbones and Backdoors in Satisfiability. In *Proceedings of the National Conference on Artificial Intelligence*, volume 3, pages 1368–1373, January 2005.

[63] Donald Knuth. *The TEXbook*. Addison-Wesley, Reading, Massachusetts, 1986.

[64] Jan Krajíček. *Proof Complexity*, volume 170 of *Encyclopedia of Mathematics and Its Applications*. Cambridge University Press, March 2019.

[65] 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.

[66] Leslie Lamport. *LATEX — A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, second edition, 1994.

[67] Massimo Lauria, Jan Elffers, Jakob Nordström, and Marc Vinyals. CNFgen: A Generator of Crafted Benchmarks. In *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT '17)*, pages 464–473, August 2017.

[68] Chunxiao Li, Jonathan Chung, Soham Mukherjee, Marc Vinyals, Noah Fleming, Antonina Kolokolova, Alice Mu, and Vijay Ganesh. On the hierarchical community structure of practical boolean formulas. In *Theory and Applications of Satisfiability Testing–SAT 2021: 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings 24*, pages 359–376. Springer, 2021.

[69] Chunxiao Li, Noah Fleming, Marc Vinyals, Toniann Pitassi, and Vijay Ganesh. Towards a complexity-theoretic understanding of restarts in sat solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 233–249. Springer, 2020.

[70] Chunxiao Li, Noah Fleming, Marc Vinyals, Toniann Pitassi, and Vijay Ganesh. Towards a complexity-theoretic understanding of restarts in sat solvers, 2020.

[71] Jia Hui Liang. *Machine Learning for SAT Solvers*. PhD thesis, University of Waterloo, Canada, 2018.

[72] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for sat solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 123–140. Springer, 2016.

[73] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning Rate Based Branching Heuristic for SAT Solvers. In *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing - SAT 2016*, pages 123–140, 2016.

[74] 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 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 123–140. Springer, 2016.

[75] Jia Hui Liang, Chanseok Oh, Minu Mathew, Ciza Thomas, Chunxiao Li, and Vijay Ganesh. Machine learning-based restart policy for cdcl sat solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 94–110. Springer, 2018.

[76] Jia Hui Liang, Pascal Poupart, Krzysztof Czarnecki, and Vijay Ganesh. An empirical study of branching heuristics through the lens of global learning rate. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 119–135. Springer, 2017.

[77] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47(4):173–180, 1993.

[78] Norbert Manthey and Jonathan Heusser. Satcoin–bitcoin mining via sat. *Proceedings of SAT Competition*, 2018:67–68, 2018.

[79] João Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 133–182. IOS Press, 2021.

[80] Joao P Marques-Silva and Karem A Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[81] Robert Mateescu. Treewidth in Industrial SAT Benchmarks. Technical Report MSR-TR-2011-22, Microsoft, February 2011.

[82] Rémi Monasson, Riccardo Zecchina, Scott Kirkpatrick, Bart Selman, and Lidror Troyansky. Determining Computational Complexity from Characteristic 'Phase Transitions'. *Nature*, 400(6740):133–137, 1999.

[83] 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*, pages 530–535. ACM, 2001.

[84] Nathan Mull, Daniel J. Fremont, and Sanjit A. Seshia. On the Hardness of SAT with Community Structure. In *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 141–159, July 2016.

[85] Nathan Mull, Shuo Pang, and Alexander Razborov. On CDCL-based proof systems with the ordered decision strategy. *arXiv preprint arXiv:1909.04135*, 2019.

[86] Saeed Nejati, Jia Hui Liang, Catherine Gebotys, Krzysztof Czarnecki, and Vijay Ganesh. Adaptive restart and cegar-based solver for inverting cryptographic hash functions. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 120–131. Springer, 2017.

[87] Saeed Nejati, Zack Newsham, Joseph Scott, Jia Hui Liang, Catherine Gebotys, Pascal Poupart, and Vijay Ganesh. A propagation rate based splitting heuristic for divide-and-conquer solvers. In *Theory and Applications of Satisfiability Testing–SAT 2017: 20th International Conference, Melbourne, VIC, Australia, August 28–September 1, 2017, Proceedings 20*, pages 251–260. Springer, 2017.

[88] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(2), February 2004.

[89] Zack Newsham, Vijay Ganesh, Sebastian Fischmeister, Gilles Audemard, and Laurent Simon. Impact of Community Structure on SAT Solver Performance. In *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, pages 252–268, 2014.

[90] Albert Oliveras, Chunxiao Li, Darryl Wu, Jonathan Chung, and Vijay Ganesh. Learning shorter redundant clauses in sdcl using maxsat. In *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

[91] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[92] Knot Pipatsrisawat and Adnan Darwiche. A new clause learning scheme for efficient unsatisfiability proofs. In *AAAI*, pages 1481–1484, 2008.

[93] Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning SAT solvers as resolution engines. *Artificial Intelligence*, 175(2):512–525, 2011.

[94] Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning SAT solvers as resolution engines. *Artif. Intell.*, 175(2):512–525, 2011.

[95] John C. Platt. Probabilistic Outputs for Support Vector Machines and Comparisons to Regularized Likelihood Methods. In *Advances in Large Margin Classifiers*, pages 61–74. MIT Press, 1999.

[96] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in sat-based formal verification. *Int. J. Softw. Tools Technol. Transf.*, 7(2):156–173, 2005.

[97] Erzsébet Ravasz, Anna Lisa Somera, Dale A Mongru, Zoltán N Oltvai, and A-L Barabási. Hierarchical Organization of Modularity in Metabolic Networks. *science*, 297(5586):1551–1555, 2002.

[98] Joseph E. Reeves, Marijn J. H. Heule, and Randal E. Bryant. Preprocessing of propagation redundant clauses. In Jasmin Blanchette, Laura Kovács, and Dirk Pattinson, editors, *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*, volume 13385 of *Lecture Notes in Computer Science*, pages 106–124. Springer, 2022.

[99] Robert Robere. Personal communication, 2018.

[100] Marko Samer and Stefan Szeider. Backdoor Trees. In *Automated Reasoning*, volume 1, pages 363–368. Springer, January 2008.

[101] Marko Samer and Stefan Szeider. Fixed-parameter tractability. *Handbook of Satisfiability*, 185:425–454, 2009.

[102] SAT. The International SAT Competition. http://www.satcompetition.org, 2022. Accessed: 2023-05-05.

[103] Bart Selman, David G Mitchell, and Hector J Levesque. Generating Hard Satisfiability Problems. *Artificial intelligence*, 81(1-2):17–29, 1996.

[104] SHARCNET. SHARCNET: Graham Cluster. https://www.sharcnet.ca/my/systems/show/114. Accessed: 2021-03-06.

[105] João P. Marques Silva and Karem A. Sakallah. Invited tutorial: Boolean satisfiability algorithms and applications in electronic design automation. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, page 3. Springer, 2000.

[106] Herbert A. Simon. The Architecture of Complexity. *Proceedings of the American Philosophical Society*, 106(6):467–482, 1962.

[107] Robert George Douglas Steel and James Hiram Torrie. *Principles and Procedures of Statistics*. McGraw-Hill, 1960.

[108] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[109] Stephan Szeider. Algorithmic Utilization of Structure in SAT Instances. Theoretical Foundations of SAT/SMT Solving Workshop at the Simons Institute for the Theory of Computing, 2021.

[110] William R Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3-4):285–294, 1933.

[111] Grigori S Tseitin. On the complexity of derivation in propositional calculus. *Automation of reasoning: 2: Classical papers on computational logic 1967–1970*, pages 466–483, 1983.

[112] Alasdair Urquhart. Hard examples for resolution. *J. ACM*, 34(1):209–219, 1987.

[113] Allen Van Gelder. Pool resolution and its relation to regular resolution and DPLL with clause learning. In *Proceedings of the 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR '05)*, volume 3835 of *Lecture Notes in Computer Science*, pages 580–594. Springer, 2005.

[114] Marc Vinyals. Hard examples for common variable decision heuristics. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI '20)*, February 2020.

[115] Ryan Williams, Carla Gomes, and Bart Selman. On the connections between backdoors, restarts, and heavy-tailedness in combinatorial search. *structure*, 23:4, 2003.

[116] Ryan Williams, Carla P. Gomes, and Bart Selman. Backdoors to typical case complexity. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI 2003)*, pages 1173–1178, 2003.

[117] Yichen Xie and Alexander Aiken. Saturn: A SAT-Based Tool for Bug Detection. In *Proceedings of the 17th International Conference on Computer Aided Verification, CAV 2005*, pages 139–143, 2005.

[118] Lin Xu, Frank. Hutter, Holger Hoos, and Kevin Leyton-Brown. Features for SAT. http://www.cs.ubc.ca/labs/beta/Projects/SATzilla/, 2012. Accessed: 2021-02.

[119] Lintao Zhang and Sharad Malik. Conflict driven learning in a quantified boolean satisfiability solver. In Lawrence T. Pileggi and Andreas Kuehlmann, editors, *Proceedings of the 2002 IEEE/ACM International Conference on Computer-aided Design, ICCAD 2002, San Jose, California, USA, November 10-14, 2002*, pages 442–449. ACM / IEEE Computer Society, 2002.

[120] Jiongzhi Zheng, Kun He, Zhuo Chen, Jianrong Zhou, and Chu-Min Li. Combining hybrid walking strategy with kissat mab, cadical, and lstech-maple. *SAT COMPETITION 2022*, page 20, 2022.

[121] Edward Zulkoski, Ruben Martins, Christoph M. Wintersteiger, Jia Hui Liang, Krzysztof Czarnecki, and Vijay Ganesh. The Effect of Structural Measures and Merges on SAT Solver Performance. In *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming*, pages 436–452, 2018.

[122] Edward Zulkoski, Ruben Martins, Christoph M. Wintersteiger, Robert Robere, Jia Hui Liang, Krzysztof Czarnecki, and Vijay Ganesh. Learning-Sensitive Backdoors with Restarts. In *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming*, pages 453–469, 2018.