Understanding and Enhancing CDCL-based SAT Solvers

by

Edward Zulkoski

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

Waterloo, Ontario, Canada, 2018

Examining Committee Membership

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

External Examiner:	Fahiem Bacchus Professor, Dept. of Computer Science, University of Toronto
Supervisor(s):	Vijay Ganesh Professor, Dept. of Computer Science, University of Waterloo Krzysztof Czarnecki Professor Dept. of Computer Science, University of Waterloo
Internal Member:	Nancy Day Professor, Dept. of Computer Science, University of Waterloo
Internal Member:	Peter van Beek Professor, Dept. of Computer Science, University of Waterloo
Internal-External Member:	Arie Gurfinkel Professor, Dept. of Electrical and Computer Engineering, University of Waterloo

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

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

Abstract

Modern conflict-driven clause-learning (CDCL) Boolean satisfiability (SAT) solvers routinely solve formulas from industrial domains with millions of variables and clauses, despite the Boolean satisfiability problem being NP-complete and widely regarded as intractable in general. At the same time, very small crafted or randomly generated formulas are often infeasible for CDCL solvers. A commonly proposed explanation is that these solvers somehow exploit the underlying structure inherent in industrial instances. A better understanding of the structure of Boolean formulas not only enables improvements to modern SAT solvers, but also lends insight as to *why* solvers perform well or poorly on certain types of instances. Even further, examining solvers through the lens of these underlying structures can help to distinguish the behavior of different solving heuristics, both in theory and practice.

The first issue we address relates to the representation of SAT formulas. A given Boolean satisfiability problem can be represented in arbitrarily many ways, and the type of encoding can have significant effects on SAT solver performance. Further, in some cases, a direct encoding to SAT may not be the best choice. We introduce a new system that integrates SAT solving with computer algebra systems (CAS) to address representation issues for several graph-theoretic problems. We use this system to improve the bounds on several finitely-verified conjectures related to graph-theoretic problems. We demonstrate how our approach is more appropriate for these problems than other off-the-shelf SAT-based tools.

For more typical SAT formulas, a better understanding of their underlying structural properties, and how they relate to SAT solving, can deepen our understanding of SAT. We perform a large-scale evaluation of many of the popular structural measures of formulas, such as community structure, treewidth, and backdoors. We investigate how these parameters correlate with CDCL solving time, and whether they can effectively be used to distinguish formulas from different domains. We demonstrate how these measures can be used as a means to understand the behavior of solvers during search. A common theme is that the solver exhibits locality during search through the lens of these underlying structures, and that the choice of solving heuristic can greatly influence this locality. We posit that this local behavior of modern SAT solvers is crucial to their performance.

The remaining contributions dive deeper into two new measures of SAT formulas. We first consider a simple measure, denoted "mergeability," which characterizes the proportion of input clauses pairs that can resolve and merge. We develop a formula generator that takes as input a seed formula, and creates a sequence of increasingly more mergeable formulas, while maintaining many of the properties of the original formula. Experiments over randomly-generated industrial-like instances suggest that mergeability strongly negatively correlates with CDCL solving time, i.e., as

the mergeability of formulas increases, the solving time decreases, particularly for unsatisfiable instances.

Our final contribution considers whether one of the aforementioned measures, namely backdoor size, is influenced by solver heuristics in theory. Starting from the notion of learning-sensitive (LS) backdoors, we consider various extensions of LS backdoors by incorporating different branching heuristics and restart policies. We introduce learning-sensitive with restarts (LSR) backdoors and show that, when backjumping is disallowed, LSR backdoors may be exponentially smaller than LS backdoors. We further demonstrate that the size of LSR backdoors are dependent on the learning scheme used during search. Finally, we present new algorithms to compute upper-bounds on LSR backdoors that intrinsically rely upon restarts, and can be computed with a single run of a SAT solver. We empirically demonstrate that this can often produce smaller backdoors than previous approaches to computing LS backdoors.

Acknowledgements

I have been very fortunate to have had the support of many incredible people during my PhD. Waterloo has been an exceptional place to study and I am grateful for the opportunity. I would like to thank the following people, without which this dissertation could not be possible:

- My supervisors Vijay Ganesh and Krzysztof Czarnecki. Thank you Vijay for our many productive conversations, and for pushing me to dive deeper into fruitful topics when I had all but given up. Thank you Krzysztof for opening countless avenues for research, and for allowing me the freedom to explore the research that truly interests me. Thank you both for kind temperament and your exceptional leadership qualities, which will serve as a shining example for me as I continue my career. Your guidance has been paramount to the creation of this thesis.
- All the faculty members that have helped me grow as a researcher throughout my time at Waterloo. In particular, thanks to Nancy Day for agreeing to be on my committee, and for the many enjoyable teaching experiences in SE212. Likewise, thank you to Peter van Beek for being on my committee, and for your feedback throughout my degree. I would also like to thank Derek Rayside for our earlier research collaboration.
- Thank you to Arie Gurfinkel and Fahiem Bacchus for agreeing to be on my committee and review my thesis.
- My collaborators Christoph Wintersteiger, Ruben Martins, and Robert Robere. Thank you Christoph for hosting my internship in Cambridge; it was truly an amazing place to work. Thank you all for the many research discussions and for remaining committed to our work despite earlier setbacks.
- My friends and colleagues in the GSD and CAR labs: Alex, Atrisha, Hari, Ian, Jimmy, Kacper, Leo, Rafael, Saeed, and Pavel. Thank you for making this often arduous journey much more enjoyable.
- The many friends, soccer teammates, and climbing buddies I have enjoyed spending time with in Waterloo: Auggy, Bogdan, Cecylia, Colin, Daniela, Danny, Eddy, Eric, Kwaku, Nick, Omar, Russell, Sean, Spencer, Steven, Steph, Stephanie, Victoria, Wes, and the countless others.
- The amazing team at Quantstamp which I have greatly enjoyed working with, in particular the east coast team which I have spent most of my time with: Alex, David, Kacper, Leo, Michael, Richard, Steven, and Vajih.

- My friends back home, who were always ready to meet up when back in town, even after my occasional disappearances into a "PhD black hole": Andrew, Dakota, Doug, and Sarah.
- My amazing undergraduate professors at Wilkes University who gave me every opportunity possible to grow as a student and make this experience at Waterloo even possible. In particular, thanks to Fred Sullivan and John Harrison (who pushed me to apply to Waterloo in the first place!). Your passion for teaching is what makes Wilkes such a great university.
- Jean Webster and Vera Korody for their efforts facilitating financial and travel logistics.
- Amy Orris, for her endless patience and support in enduring this long journey.
- Last, but certainly not least, I would like to thank all of my family, particularly my parents Terese and Ed Zulkoski, and my brother Eric. Thank you for your unwavering support throughout my degree.

Dedication

To my family.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern conflict-driven clause-learning (CDCL) satisfiability (SAT) solvers are routinely used to solve formulas with hundreds of thousands of variables and millions of clauses, despite the Boolean satisfiability problem being NP-complete. Nonetheless, small real-world, random, or crafted SAT instances such as the pigeonhole problem, cryptographic instances, and certain mathematical instances are difficult for CDCL solvers [Biere, 2016, Heule et al., 2016b, Konev and Lisitsa, 2014]. The feasibility of SAT solving large real-world instances has therefore perplexed both theoreticians and solver developers alike. A commonly proposed explanation is that the solver can exploit the underlying structure of problems found in practice. Even further, many problems which could benefit from SAT solver's reasoning capabilities cannot be succinctly mapped to Boolean formulas. This is in some sense a *representational* issue: certain constraints from higher-level domains than Boolean logic require large size-blowups when converted to Boolean formulas, dooming the SAT solver before it even starts. The goal of this thesis is to build upon CDCL SAT solvers to handle certain types of problems which cannot be succinctly represented as Boolean formulas, and for more typical Boolean formulas, to better understand how the structural properties of these formulas relate to CDCL SAT solving.

Thesis Statement: *CDCL-based solvers can be extended to solve certain types of formulas derived from hard combinatorial problems, through abstraction of hard predicates. For more general classes of formulas, analyzing how underlying structural properties of formulas relate to solver behaviour can improve our understanding of CDCL SAT solvers, both in theory and practice.*

1.1 CDCL+CAS for Solving Hard Combinatorial Problems

While Boolean satisfiability solvers are capable of handling constraints stemming from a large variety of domains, there are many problems that could benefit from the search capabilities of SAT, but are either not easily expressed in, or require large blowups in problem size when converted to Boolean logic. Extensions of SAT solvers such as modern satisfiability modulo theories (SMT) solvers (e.g. Z3 [De Moura and Bjørner, 2008], CVC4 [Barrett et al., 2011], STP [Ganesh and Dill, 2007], and VERIT [Bouton et al., 2009]) contain efficient decision procedures for a variety of first-order theories, such as uninterpreted functions, quantified linear integer arithmetic, bitvectors, and arrays. However, even with the expressiveness of SMT, many constraints, particularly those stemming from mathematical domains such as graph theory, topology, algebra, or number theory are non-trivial to solve using today's state-of-the-art SAT and SMT solvers. Computer algebra systems (e.g., MAPLE [Char et al., 1986], MATHEMATICA [Wolfram, 1999], MAGMA [Bosma et al., 1997] and SAGE [Stein and Etal., 2010]), on the other hand, are powerful tools that have been used for decades by mathematicians to perform symbolic computation over problems in graph theory, topology, algebra, number theory, etc. However, when applied to prove or disprove a certain statement, computer algebra systems (CAS) lack the search capabilities of SAT/SMT solvers, which are a central aspect of the latter tools.

For our first contribution, we present a method and a prototype tool, called MATHCHECK, that combines the search capability of SAT solvers with powerful domain knowledge of CAS systems The tool MATHCHECK can solve problems that are too difficult or inefficient to encode as SAT problems. MATHCHECK can be used by mathematicians to finitely check or find counterexamples to open conjectures. It can also be used by engineers who want to readily leverage the joint capabilities of both CAS systems and SAT solvers to model and solve problems that are otherwise too difficult with either class of tools alone.

The key concept behind MATHCHECK is that it embeds the functionality of a computer algebra system (CAS) within the inner loop of a CDCL SAT solver. Computer algebra systems contain state-of-the-art algorithms from a broad range of mathematical areas, many of which can be used as subroutines to easily encode predicates relevant both in mathematics and engineering. The users of MATHCHECK write predicates in the language of the CAS, which then interacts with the SAT solver through a controlled SAT+CAS interface. The user's goal is to finitely check or find counterexamples to a Boolean combination of predicates (somewhat akin to a quantifier-free SMT formula). The SAT solver searches for counterexamples in the domain over which the predicates are defined, and invokes the CAS to learn clauses that help cut down the search space (akin to the "T" in DPLL(T)).

In this work, we focus on constraints from the domain of graph theory, although our approach is

equally applicable to other areas of mathematics.[1] Constraints in graph theory such as connectivity, Hamiltonicity, acyclicity, etc. are non-trivial to encode with standard solvers, and can lend themselves to many possible encodings of widely ranging performance [Velev and Gao, 2009]. We present two case studies where we finitely verify and extend known results on two open conjectures over hypercubes. We believe that the method described in this work is a step in the right direction towards making SAT/SMT solvers useful to a broader class of mathematicians and engineers than before.

1.2 Relating Structural Properties to CDCL Solving

For more traditional Boolean formulas which can be handled by standard SAT solvers, an age-old problem is to try to understand what properties of the formula relate to SAT solving and performance. One of the earliest examples of this goes back to the phase transition phenomenon observed for randomly-generated SAT instances in the late 1990's [Coarfa et al., 2000, Monasson et al., 1999, Selman et al., 1996]. It was empirically shown that when the clause/variable ratio is approximately 4.27, the fraction of satisfiable random 3SAT instances is approximately 50%, and that these instances tend to be the most difficult for solvers.

However, for more general classes of instances, particularly those from derived from industrial settings, there is no clear transition between satisfiable and unsatisfiable instances. Many alternative characterizations of formulas have been proposed in an attempt to explain why SAT solvers perform so well from instances in industrial domains. Among the most prominent characterizations, *backdoors* constitute small sets of variables such that, if correctly assigned, then the problem becomes easy [Williams et al., 2003a]. It was suggested and empirically shown that for many industrial instances, there often exists very small backdoors which include only a fraction of the total variables [Kilby et al., 2005, Li and Van Beek, 2011]. As another example, the community structure of a Boolean formula characterizes graphical abstraction of the formula by partitioning variables (represented as nodes in the graph) into highly modular partitions [Ansótegui et al., 2012]. It was shown that properties of this graph abstraction correlate moderately with CDCL performance [Newsham et al., 2014]. Other measures have been considered such as treewidth [Mateescu, 2011], backbones [Monasson et al., 1999], and many extensions of backdoors [Ruan et al., 2004, Samer and Szeider, 2008, Dilkina et al., 2009b, Ganian et al., 2017].

In order to assess how these measures relate to CDCL performance and typical industrial instances, we first construct regression models over the measures to assess how well they correlate

[1] Extensions of our work have considered conjectures regarding Hadamard and Williamson matrices [Bright et al., 2016a, Bright, 2017].

with solving time. Measures that better correlate with solving time should be a good starting point for further investigation. We also investigate which parameters easily distinguish the different classes of instances (e.g. industrial vs random instances).

Although several such studies of these parameters have been performed in isolation, a comprehensive comparison has not been performed between them. A primary reason for this is that most of these parameters are difficult to compute – often NP-hard – and many take longer to compute than solving the original formula. Further, certain parameters such as weak backdoor size are only applicable to satisfiable instances [Williams et al., 2003a]. Hence, such parameters have often been evaluated on incomparable benchmark sets, making a proper comparison between them difficult. We correct this issue in our study by focusing on instances found in previous SAT competitions, specifically from the application, crafted, and agile tracks [SAT, 2017], and construct regression models between structural measures and solving time. These instances are used to evaluate state-of-the-art SAT solvers on a yearly basis. Application instances are derived from a wide variety of sources and can be considered a small sample of the types of SAT instances found in practice, such as from verification domains. Crafted instances mostly contain encodings of combinatorial/mathematical properties, such as the pigeon-hole principle or pebbling formulas. While many of these instances are much smaller than industrial instances, they are often very hard for CDCL solvers. The agile track evaluates solvers on bit-blasted quantifier-free bit-vector instances generated from the whitebox fuzz tester SAGE [Godefroid et al., 2008]. In total, we consider approximately 1200 application instances, 800 crafted instances, and 5000 Agile instances. While no single measure seems to highly correlate with solving time, combinations of small sets of them do produce moderate to strong correlations.

Going beyond our correlation results, we show how some of these structural parameters can be used as a *lens* to analyze the effects of various solving heuristics. Our experiments focus on branching heuristics (VSIDS [Moskewicz et al., 2001], LRB [Liang et al., 2016b], and random) and restart policies (Luby [Luby et al., 1993], restarting after every conflict, and never restarting). First, we show that the proofs that the solver finds when restarting after every conflict are significantly more local than the other restart policies. In [Zulkoski et al., 2017c], we introduced and formalized the concept of learning-sensitive with restarts (LSR) backdoors, which extend learning-sensitive (LS) backdoors [Dilkina et al., 2009a]. Essentially, LSR backdoors measure the minimal number of unique variables that the solver much branch upon to solve the instance. We showed that the set of variables in the learned clauses used in the proof constitute a (not necessarily minimal) LSR backdoor. We use this notion to measure the locality of the proofs found by the solver.

Second, various works have considered either pre-computing backbone literals, or using heuristics to identify more backbone literals in the hopes of improving solver performance [Batory, 2005, Lonsing and Biere, 2011, Manolios and Papavasileiou, 2011]. We analyze how

much work the solver could have avoided with *a priori* knowledge of the backbone, by considering how often the solver learns "backbone-subsumed" clauses. Finally, we reconsider the community-based spatial locality experiments from [Liang et al., 2015b] with our expanded set of considered heuristics and new benchmarks.

1.3 Two New Characterizations of SAT Formulas

Our final contributions delve deeper into two specific measures which offer further characterization of differing solving heuristics, and further explanation of SAT solver performance. We first consider a simple measure, which we call "mergeability," which quantifies how many pairs of input clauses are mergeable. Two clauses are mergeable if they resolve and share a common literal. Merge resolutions are particularly important, as they allow the resolvent clause to be smaller than the two clauses being resolved. We describe an algorithm which takes a formula, and produces a series of increasingly more mergeable formulas, while retaining many properties of the original instance. We experiment over a set of randomly-generated industrial-like instances, and show that as the number of merges increase, the solving time tends to decrease, particularly for unsatisfiable instances.

Finally, we introduce *learning sensitive with restarts (LSR) backdoors*, an extension of *learning sensitive (LS) backdoors* [Dilkina et al., 2009b]. LS backdoors extend more traditional backdoor definitions by allowing clause learning to occur while exploring the search space of backdoor variables. LSR backdoors further allow restarts during search. We first demonstrate separations of several solving heuristics through the lens of LSR backdoors. Our main result is an exponential separation between LSR and LS backdoors (when backjumping is disallowed). Determining whether or not restarts add significant power to CDCL SAT solvers in full generality remains a major and important open problem. We hope that our work will be a useful step toward tackling this problem.

We further show that different learning policies may have exponential separations as well. We show that for certain classes of formulas, LSR backdoors with the *first unique implication point* learning policy [Marques-Silva and Sakallah, 1999], the most commonly used learning policy in practice, may have exponentially smaller minimum LSR backdoors than if the *decision learning* policy [Zhang et al., 2001] is used. Among other results, we develop several heuristic algorithms to overapproximate the size of LSR backdoors in practice, and demonstrate that rapid restart policies tend to lead to smaller LSR backdoors.

1.4 Supporting Code Contributions

This dissertation is partially based on the following code:

- We developed a SAT+CAS system, called MATHCHECK, which combines the SAGE CAS [Stein and Etal., 2010] with the Glucose SAT solver [Audemard and Simon, 2009], available at [Zulkoski and Ganesh, 2015]. The system contains a set of pre-implemented graph theoretic constraints and can be easily extended to handle new constraints. The tool exposes many of the APIs of SAGE's graph theoretic libraries to facilitate creating constraints.
- We developed a tool called *LaSeR* for computing overapproximations of LSR backdoors [Zulkoski et al., 2017b]. The tool is further capable of verifying backdoors, and computing minimal backdoors (for small crafted instances).
- Additional supporting code for correlation experiments and further relating solver behaviour to structural measures is available at [Zulkoski, 2017] and [Zulkoski and Ganesh, 2017].

1.5 Supporting Publications

This dissertation contains material from the following publications:

- [Zulkoski et al., 2015] Edward Zulkoski, Vijay Ganesh, and Krzsztof Czarnecki. *MathCheck: A math assistant via a combination of computer algebra systems and SAT solvers.* In International Conference on Automated Deduction (CADE) 2015.
- [Liang et al., 2015b] Jia Hui Liang, Vijay Ganesh, Edward Zulkoski, Atulan Zaman, and Krzysztof Czarnecki. *Understanding VSIDS branching heuristics in conflict-driven clause-learning SAT solvers.* In Haifa Verification Conference (HVC) 2015.
- [Zulkoski et al., 2016] Edward Zulkoski, Vijay Ganesh, and Krzsztof Czarnecki. *MathCheck: A math assistant via a combination of computer algebra systems and SAT solvers.* In International Joint Conference on Artificial Intelligence – Sister Conferences Best Paper Track (IJCAI) 2016.
- [Zulkoski et al., 2017a] Edward Zulkoski, Curtis Bright, Albert Heinle, Ilias Kotsireas, Krzysztof Czarnecki, and Vijay Ganesh. *Combining SAT solvers with computer algebra systems to verify combinatorial conjectures.* In Journal of Automated Reasoning (JAR) 2017.

- [Zulkoski et al., 2017d] Edward Zulkoski, Ruben Martins, Christoph Wintersteiger, Robert Robere, Jia Hui Liang, Krzysztof Czarnecki, and Vijay Ganesh. *Empirically Relating Complexity-theoretic Parameters with SAT Solver Performance.* In Pragmatics of Constraint Reasoning (PoCR) 2017.

- [Zulkoski et al., 2018a] Edward Zulkoski, Ruben Martins, Christoph Wintersteiger, Jia Hui Liang, Krzysztof Czarnecki, and Vijay Ganesh. *The Effect of Structural Measures and Merges on SAT Solver Performance.* In International Conference on Principles and Practice of Constraint Programming (CP) 2018.

- [Zulkoski et al., 2018b] Edward Zulkoski, Ruben Martins, Christoph Wintersteiger, Robert Robere, Jia Hui Liang, Krzysztof Czarnecki, and Vijay Ganesh. *Learning Sensitive Backdoors with Restarts.* In International Conference on Principles and Practice of Constraint Programming (CP) 2018.

Chapter 2

Background

In this chapter, we introduce definitions and preliminary material for the following sections.

2.1 The Boolean Satisfiability Problem

A *Boolean variable* is a variable that can be assigned either `true` or `false` (sometimes encoded as 1 or 0, respectively). A *literal* is a Boolean variable v or its negation, denoted $\neg v$. A *clause* over the literals $l_1, \ldots, l_n$ is a disjunction of literals, denoted $l_1 \vee \ldots \vee l_n$. A formula is in *conjunctive normal form (CNF)* if it is a conjunction of disjunction of literals. We assume all formulas are in CNF. We often refer to a clause as a set of literals, and a CNF formula as a set of clauses. Any formula that is not in CNF can be converted to CNF; this is typically done through the Tseitin-transformation [Tseitin, 1968].

Let F be a CNF formula. We typically denote the full set of variables that appear in F as *vars(F)*, and the set of clauses that appear in F as *clauses(F)*. An *assignment* α maps a set of variables V to values. If V is the full set of variables in some formula, then α is a *complete assignment*. A literal v (resp. $\neg v$) is satisfied by α if, and only if it assigns v to `true` (resp. `false`). A clause C is satisfied by α if, and only if there exists some literal $l \in C$ such that α satisfies l. We use the notion $F[\alpha]$ to mean the formula F gets simplified by α by removing all satisfied clauses, and removing literals that are falsified by α in the remaining clauses. A formula F is *satisfiable* if, and only if there exists a complete assignment α to vars(F) such that all clauses in F are satisfied. We call such a satisfying assignment a *model* for F. If no model exists, then the formula is *unsatisfiable*. The *Boolean Satisfiability Problem (SAT)* is to determine whether any given formula F is satisfiable.

Algorithm 1 CDCL SAT Solver (adapted from [Pipatsrisawat and Darwiche, 2011])

input: CNF Formula F
output: SAT or UNSAT
$T \leftarrow \langle\rangle$ ▷ trail of decision literals and propagations
$L \leftarrow \emptyset$ ▷ the set of learned clauses
while `true` **do**
    $T \leftarrow \mathit{unitPropagate}(F, L, T)$
    if (F, L, T) is in conflict **then** ▷ unit propagation detects the empty clause
        if $T = \langle\rangle$ **then** ▷ the conflict is at decision level 0
            return UNSAT
        $c \leftarrow \mathit{analyzeConflict}(F, L, T)$ ▷ derive a conflict clause
        $m \leftarrow \mathit{assertionLevel}(c)$
        $T \leftarrow T_m$ ▷ clear any decisions/propagations after decision level m
        $L = L \cup \{c\}$
    else ▷ no conflict
        if time to restart **then**
            $T \leftarrow \langle\rangle$ ▷ clear the trail
        l = *pickBranchLiteral*() ▷ $l \notin T$ and $\neg l \notin T$
        if $l =$ null **then** ▷ all variables have already been assigned
            return SAT
        $T \leftarrow T, l$ ▷ add the decision literal to the trail

2.2 Conflict-driven Clause-learning SAT Solvers

We only highlight the main aspects of the conflict-driven clause-learning (CDCL) SAT solvers. For an extended overview of CDCL SAT solvers, we refer to [Biere et al., 2009a]. Pseudocode for a basic CDCL solver is presented in Algorithm 1. CDCL solvers essentially work by traversing the space of variable assignments through heuristically assigning values to variables, and pruning the search space with learned clauses whenever an unsatisfying state is reached during search. The solver take as input a CNF formula F, and return SAT (plus a model if F is satisfiable), or UNSAT (optionally with a proof of unsatisfiability). The *trail* refers to the sequence of variable assignments, in the order they have been assigned, at any given point during the run of a solver. Learned clauses are derived by analyzing the *implication graph*, which represents which decisions and propagations that led to a conflict. Unless otherwise stated, we assume the *first unique implication point (`1UIP`)* clause learning scheme throughout, which is the most common in practice [Moskewicz et al., 2001]. Importantly, learned clauses (also referred to as *conflict*

clauses) are implied by the original formula F, so including them in the set of clauses does not affect satisfiability.

The solver repeats the following steps until completion. Unit propagation adds any literals to the trail that are implied by any clause (Line 6). A literal is implied if there exists a clause such that all but one literal is falsified by the current trail, and the last literal is unassigned. In this case, the remaining literal must be set to `true`, and is added to the trail as an *implied literal*. Literals that are instead added to the trail by branching (as in Lines 17-20) are called *decision variables*. The *decision level* of a literal on the trail is defined as the number of decision variables that occur before it in the trail, including the literal itself if it is a decision.

If the solver detects that all literals in a clause are set to `false`, it has reached a conflicting state (Line 7). If the conflict occurs when there are no decision literals in the trail, then the formula is unsatisfiable, and we are done (Lines 8-9). Otherwise, conflict analysis derives a new learned clause c (Line 10). The assertion level m is defined as the second highest decision level of all literals in the clause (zero if the clause is unit). The solver *backjumps* to decision level m by clearing any literals in the trail with level greater than m (thus escaping the conflicting state), adds c to the set of learned clauses, and continues (Lines 11-13).

Otherwise, the solver is not in conflict, and the solver must continue to explore the search-space in one of two ways. First, the solver may choose to *restart* by clearing the trail while retaining all learned clauses (Lines 15-16; "time to restart" is usually defined by a specified number of conflicts). This may help to prevent the solver from "getting stuck" in unfavorable regions of the search space. Otherwise, the solver heuristically chooses to branch upon a literal. If all variables are already assigned, then the formula must be satisfiable (Line 18-19), otherwise we add the literal to the trail and continue.

Note that in the case of unsatisfiable formulas, although many clauses may be learned, only a small portion may be actually used to construct to proof of unsatisfiability. We formally define "useful clauses" as the ones needed for the proof:

Definition 1 (Useful clauses)**.** Let P be a proof of unsatisfiability constructed by the SAT solver represented as a graph G, such that nodes represent clauses, input clauses have no incoming edges, and an edge exists from C_1 to C_2 *iff* the clause C_1 was in the implication graph used to derive C_2. (Additional edges are needed to account for extra components of real-world solvers, such as clause minimization.) The final node added to the graph is the empty clause E. Then, if we reverse all edges in the graph, the *useful clauses* correspond to the set of nodes reachable from E.

2.3 Parameterizations of SAT Formulas

We discuss several measures and parameterizations which allow us to characterize [classes of] SAT formulas, beyond simple properties such as the number of variables or clauses.

2.3.1 Backdoor-related Concepts

A *strong backdoor* of a formula F, as introduced by Williams et al. [Williams et al., 2003a], is intuitively a set of variables B such that for any assignment $a_B : B \mapsto \{T, F\}$, the simplified formula $F[a_B]$ can be solved in polynomial-time. In order to further formalize this concept, we must introduce the notion of a subsolver:

Definition 2 (Subsolver [Williams et al., 2003a])**.** A subsolver S is an algorithm such that for any formula F, the following hold:

1. (Trichotomy) S either rejects F or correctly determines F to be satisfiable or unsatisfiable.
2. (Efficiency) S runs in polynomial time.
3. (Trivial solvability) S determines if F is trivially true, i.e., it contains no clauses or the empty clause.
4. (Self-reducibility) If S determines F, then for any variable x and value ε, S determines $F[\varepsilon/x]$.

Intuitively, a subsolver can be thought of as an incomplete solver that can solve some fragment of Boolean logic. Example subsolvers include so called *syntactic* solvers such as a 2-CNF solver, or more *dynamic/semantic* solvers such as a unit propagation (UP) algorithm. For a comprehensive overview of subsolvers see [Dilkina et al., 2014]. In this work, we only focus on UP as our subsolver, as this is a main subroutine that CDCL SAT solvers implement in practice.

We can now introduce both *strong* and *weak* backdoors, which we collectively refer to as *traditional* backdoors.

Definition 3 (Strong backdoor [Williams et al., 2003a])**.** A set of variables $B \subseteq vars(F)$ is a strong backdoor with respect to a subsolver S if for every assignment $a_B : B \to \{T, F\}$, S determines $F[a_B]$ to be satisfiable or unsatisfiable.

Definition 4 (Weak backdoor [Williams et al., 2003a])**.** A set of variables $B \subseteq vars(F)$ is a weak backdoor with respect to a subsolver S if there exists an assignment $a_B : B \to \{T, F\}$ such that S determines $F[a_B]$ to be satisfiable.

Example 1. Consider the following formula F and assignment α.

$$\begin{aligned} F &:= (x \vee y \vee z) \wedge (\neg x \vee w) \wedge (\neg w \vee z) \\ \alpha &:= \{x \mapsto 1\} \\ \beta &:= \{x \mapsto 0\} \\ F[\alpha] &:= (w) \wedge (\neg w \vee z) \\ F[\beta] &:= (y \vee z) \wedge (\neg w \vee z) \end{aligned}$$

After simplifying the formula with α, unit propagation can determine that the formula $F[\alpha]$ is satisfiable by setting w to `true` and then z to `true`. Thus, the set $\{x\}$ constitutes a weak-UP backdoor for F. However, when setting x to `false` (assignment β), unit propagation cannot determine the satisfiability of $F[\beta]$, so $\{x\}$ is not a strong-UP backdoor for F.

The *backbone* of a SAT instance is the set of variables such that all models of the instance contain the same polarity of the variable [Monasson et al., 1999]. Note that weak backdoors and backbones are implicitly only defined over satisfiable instances. Further, while the backbone of an instance is unique, many strong and weak backdoors may exist; we typically try to find the smallest weak backdoors possible.

Backdoors were further extended to allow clause-learning to occur while exploring the search space of the backdoor:

Definition 5 (Learning-sensitive (LS) backdoor [Dilkina et al., 2009b])**.** A set of variables $B \subseteq vars(F)$ is an LS backdoor with respect to a subsolver S if there exists a search tree exploration order such that a clause-learning SAT solver branching only on variables in B, and with S as the subsolver at the leaves of the search tree, can determine the satisfiability of F.

In [Dilkina et al., 2009b], the authors demonstrate that for certain classes of formulas the smallest LS backdoors, with UP as the subsolver can be exponentially smaller than the smallest strong-UP backdoor. In Chapter 6, we extend LS backdoor to allow restarts.

2.3.2 Graph Abstractions

All graphs discussed in this thesis are undirected. The *variable incidence graph (VIG)* of a CNF formula F is defined as follows: vertices of the graph are the variables in the formula. For every

clause $c \in F$ we have an edge between each pair of variables in c. In other words, each clause corresponds to a clique between its variables. The weight of an edge is $\frac{1}{|c|-1}$ where $|c|$ is the length of the clause. The VIG does not distinguish between positive and negative occurrences of variables. We combine all edges between each pair of vertices into one weighted edge by summing the weights. More precisely, the VIG of a CNF formula F is a weighted graph defined as follows: a set of vertices $V = vars(F)$, a set of edges $E = \{xy \mid x,y \in c \in F\}$, and an edge weight function $w(xy) = \sum_{x,y \in c \in F} \frac{1}{|c|-1}$.

For the *incidence graph* of a CNF formula, we introduce one vertex for every variable, and one vertex for every clause. An edge exists between a "variable vertex" and a "clause vertex" exactly when the corresponding variable occurs in the corresponding clause. No edges exists from between two variable vertices, nor between two clause vertices. The sets of variable vertices and clause vertices therefore form a bipartition of the incidence graph.

The community structure of a graph is a partition of the vertices into communities, such that there are more intracommunity edges than intercommunity edges. The *modularity* or Q value measures the quality of the community structure of the graph. The Q value ranges from $[-1/2, 1)$, where values near 1 indicate that the communities are highly separable and the graph intuitively has a *better* community structure.

2.3.3 Other Measures

We define several further measures which describe the proportion of input clauses that can resolve or merge. We discuss further in Chapter 5. Two clauses c_1 and c_2 are *resolvable* if there exists a variable v such that $v \in c_1$ and $\neg v \in c_2$. We say that the clauses are *mergeable* if they are resolvable, and there also exists some literal l such that $l \in c_1$ and $l \in c_2$. We consider two measures of formulas that quantify basic semantic properties of the input. Let C be the number of clauses in the formula. Let R be the number of resolvable pairs of clauses, and M be the number of mergeable clauses, such that if a pair merges n times, M is incremented n times. Then the *resolvability* of the input formula is R/C^2 and the *mergeability* is M/C^2.

2.4 Statistical Concepts

We perform several statistical tests to assess how various measures of SAT formulas relate to solving time. *Linear regression* models the scalar dependence of one dependent variable y (typically time) to several independent variables X. The R^2 assesses the percentage of change in the dependent variable that is explained by the independent variables, according to the model. The

value ranges between $[0,1]$, where higher indicates a better correlation. Let m be a linear model over features X that predicts y, and (X_{obs}, y_{obs}) be an observed data point. Then $m(X_{obs}) = y_{pred}$ is the predicted value of y given the model. The *residual* measures the difference between observed and predicted values: $y_{obs} - y_{pred}$. When assessing a model, we consider the *confidence level* (corresponding to p-values), that the feature is significant to the regression. Confidence values are measured as a percentage; for the scope of this work, we consider values over 99% as very significant, values between 99% and 95% are significant, and values below 95% are insignificant.

Given features X, a *classifier* attempts to determine which value from a set of categories that the observation belongs. In our context, we use classifiers to predict the domain from which a SAT formula was derived (e.g. bounded model checking or cryptographic formulas). We typically perform 10-fold *cross-validation* to assess our results. First, the set of observations are randomly divided into 10 equal partitions. A model is trained using 9 of the partitions (i.e. 90% of the observations), and is tested by predicting the categories of the remaining 10% of observations. This process is repeated 10 times by changing the testing set to a new partition. The average correct classification rate is reported.

2.5 Graph Theory Preliminaries

For the purposes of the case studies in Chapter 3, we introduce several graph theoretic concepts, unrelated to the abstractions of SAT formulas above. We denote a graph $G = \langle V, E \rangle$ as a set of vertices V and edges E, where an edge e_{ij} connects the pair of vertices v_i and v_j. The *order* of a graph is the number of vertices it contains. For a given vertex v, we denote its *neighbors* – vertices that share an edge with v – as $N(v)$.

The hypercube of dimension d, denoted Q_d, consists of 2^d vertices and $2^{d-1} \cdot d$ edges, and can be constructed in the following way (depicted in Figure 3.2a): label each vertex with a unique binary string of length d, and connect two vertices with an edge if and only if the Hamming distance of their labels is 1. A *matching* of a graph is a subset of its edges that mutually share no vertices. A vertex is *matched* (by a matching) if it is incident to an edge in the matching, else it is *unmatched*. A *maximal matching* M is a matching such that adding any additional edge to M violates the matching property. A *perfect matching* (resp. *imperfect matching*) M is a matching such that all (resp. not all) vertices in the graph are incident with an edge in M. A *forbidden matching* is a matching such that some unmatched vertex v exists and every $v' \in N(v)$ is matched. Intuitively, no superset of the matching can match v. Vertices in Q_d are *antipodal* if their binary strings differ in all positions (i.e., opposite "corners" of the cube). Edges e_{ij} and e_{kl} are antipodal if $\{v_i, v_k\}$ and $\{v_j, v_l\}$ are pairs of antipodal vertices. A *2-edge-coloring* of a graph is a labeling of

the edges with either red or blue. A 2-edge-coloring is *edge-antipodal* if the color of every edge differs from the color of the edge antipodal to it.

A symmetry/automorphism of a graph is a permutation of its vertices that preserves edges and non-edges. The set of all automorphisms of a graph is called its automorphism group.

Chapter 3

Combining CDCL Solvers with Computer Algebra Systems to Verify Combinatorial Conjectures

In this chapter, we present a method and a prototype tool, called MATHCHECK, that combines the search capability of SAT solvers with powerful domain knowledge of CAS systems (i.e., a toolbox of algorithms to solve a broad range of mathematical problems). The SAT+CAS tool can solve problems that are too difficult or inefficient to encode as SAT problems. The tool can be used by mathematicians to finitely check or find counterexamples to open conjectures. It can also be used by engineers who want to readily leverage the joint capabilities of both CAS systems and SAT solvers to model and solve problems that are otherwise too difficult with either class of tools alone.

The key concept behind MATHCHECK is that it embeds the functionality of a CAS within the inner loop of a CDCL SAT solver. Computer algebra systems contain state-of-the-art algorithms from a broad range of mathematical areas, many of which can be used as subroutines to easily encode predicates relevant both in mathematics and engineering. The users of MATHCHECK write predicates in the language of the CAS, which then interacts with the SAT solver through a controlled SAT+CAS interface. The user's goal is to finitely check or find counterexamples to a Boolean combination of predicates (somewhat akin to a quantifier-free SMT formula). The SAT solver searches for counterexamples in the domain over which the predicates are defined, and invokes the CAS to learn clauses that help cut down the search space (akin to the "T" in DPLL(T)).

In this chapter, we focus on constraints from the domain of graph theory, although our approach

is equally applicable to other areas of mathematics (cf. [Bright, 2017]). Constraints in graph theory such as connectivity, Hamiltonicity, acyclicity, etc. are non-trivial to encode with standard solvers, and can lend themselves to many possible encodings of widely ranging performance [Velev and Gao, 2009]. We believe that the method described here is a step in the right direction towards making SAT/SMT solvers useful to a broader class of mathematicians and engineers than before.

Most CAS's additionally support methods for computing symmetries of a group and automorphisms of graph objects, sometimes via interfacing fast graph automorphism tools such as SAUCY [Darga et al., 2008] or BLISS [Junttila and Kaski, 2007]. Symmetry breaking has been applied to SAT instances through tools such as SHATTER [Aloul et al., 2003], which converts the conjunctive normal form (CNF) of the input to a graph automorphism problem that is then solved with an off-the-shelf tool such as SAUCY. We use these methods to define symmetry breaking routines for our graph theoretic case studies, which significantly reduce solving times.

While we believe that our method is probably the first such combination of SAT+CAS systems, there has been previous work in attempting to extend SAT solvers with graph reasoning [Dooms et al., 2005, Gebser et al., 2014, Soh et al., 2014]. These works can loosely be divided into two categories: constraint-specific extensions, and general graph encodings. As an example of the first case, efficient SAT-based solvers have been designed to ensure that synthesized graphs contain no cycles [Gebser et al., 2014]. In [Soh et al., 2014], Hamiltonicity checks are reduced to *native* Boolean cardinality constraints and lazy connectivity constraints. While more efficient than standard encodings of acyclicity and Hamiltonicity constraints, these approaches lack generality. On the other hand, approaches such as in CP(Graph) [Dooms et al., 2005], a constraint satisfaction problem (CSP) solver extension, encode a core set of graph operations with which complicated predicates (such as Hamiltonicity) can be expressed. *Global constraints* [Dooms et al., 2005] can be tailored to handle predicate-specific optimizations. Although it can be non-trivial to efficiently encode global constraints, previous work has defined efficient procedures which enforce graph constraints, such as connectivity, incrementally during search [Holm et al., 2001]. Our approach is more general than the above approaches, because CAS systems are not restricted to graph theory. One might also consider a general SMT theory-plugin for graph theory. However given the diverse array of predicates and functions within the domain, a monolithic theory-plugin (other than a CAS system) seems impractical at this time.

Main Contributions:

Analysis of a SAT+CAS Combination Method and the MATHCHECK tool. In Section 3.1, we present a method and tool that combines a CAS with SAT, denoted as SAT+CAS, facilitating the creation of user-defined CAS predicates. Such tools can be used by mathematicians to finitely search or find counterexamples to universal sentences in the language of the underlying CAS.

MATHCHECK allows users to easily specify and solve complex combinatorial questions using the simple interface provided. The system can easily be extended to other domains, although we currently focus on problems coming from graph-theory.

Results on Two Open Graph-Theoretic Conjectures over Hypercubes. In Section 3.2, we use our system to extend results on two long-standing open conjectures related to hypercubes. Conjecture 1 states that any matching of any d-dimensional hypercube can extend to a Hamiltonian cycle. Conjecture 2 states that given an edge-antipodal coloring of a hypercube, there always exists a monochromatic path between two antipodal vertices. Previous results have shown Conjecture 1 (resp. Conjecture 2) true up to $d = 4$ [Fink, 2007] (resp. $d = 5$ [Feder and Subi, 2013]); we extend these two conjectures to $d = 5$ (resp. $d = 6$). We discuss symmetry breaking optimizations, in which we learn many symmetric clauses during solving, which result in an order of magnitude performance improvement for MATHCHECK on the two case studies.

Performance Analysis of MATHCHECK. In Section 3.3, we provide detailed performance analysis of MATHCHECK in terms of how much search space reduction is achieved relative to finite brute-force search, as well as how much time is consumed by each component of the system. Improvements from symmetry breaking techniques are also discussed. We additionally compare MATHCHECK to ALLOY*, a higher-order relational logic solver built on top of a SAT solver on our two graph-theoretic case studies. We chose to compare against ALLOY* as it was 1) SAT-based; and 2) expressive enough to support our two graph theoretic case studies. Results over the two case studies favor MATHCHECK.

Verification of Results. We provide details on the techniques used to check the results of our case studies in Section 3.4. In addition to checking that the input formula to the SAT solver and its output are correct, we must also ensure that the learned clauses generated by the CAS follow from the input specification. We discuss the analyses we performed and certificates generated by our tool in order to check its correctness after solving.

3.1 SAT+CAS Combination Architecture

This section describes the combination architecture of a CAS system with a SAT solver, the method underpinning the MATHCHECK tool. Figure 3.1 provides a schematic of MATHCHECK, with a more detailed description in Section 3.1.1. The key idea behind such combinations is that the CAS system is integrated in the inner loop of a conflict-driven clause-learning SAT solver, akin to how a theory solver T is integrated into a DPLL(T) system [Nieuwenhuis et al., 2004]. MATHCHECK allows the user to define predicates in the language of CAS that express some mathematical conjecture. The input mathematical conjecture can be expressed as a set of *assertions* and *queries*,

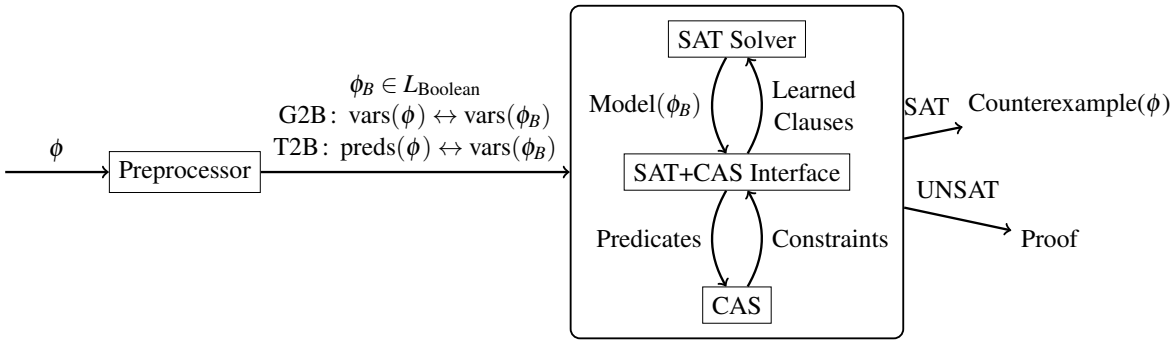


Figure 3.1: High-level overview of the MATHCHECK architecture, which is similar to DPLL(T)-style SMT solvers. MATHCHECK takes as input a formula ϕ over fragments of mathematics supported by the underlying CAS system, and produces either a counterexample or a proof that no counterexample exists.

such that a satisfying assignment to the conjunction of the assertions and *negated* queries constitute a counterexample to the conjecture. We refer to this conjunction simply as the input formula in the remainder of the paper. First, the formula is translated into Boolean constraints that describes the set of structures (e.g., graphs or numbers) referred to in the conjecture. Second, the SAT solver enumerates these structures in an attempt to counterexample the input conjecture.

The solver, solving each generated SAT instances, routinely queries the CAS system during its search to learn clauses (akin to callback plugins in programmatic SAT solvers [Ganesh et al., 2012] or theory plugins in DPLL(T) [Nieuwenhuis et al., 2004]). Clauses thus learned can dramatically cut down the search space of the SAT solver.

Combining the solver with CAS extends each of the individual tools in the following ways. First, off-the-shelf SAT (or SMT) solvers contain efficient search techniques and decision procedures, but lack the expressiveness to easily encode many complex mathematical predicates. Even if a problem can be easily reduced to SAT/SMT, the choice of encoding can be very important in terms of performance, which is typically non-trivial to determine, especially for non-experts on solvers. For example, Velev et al. [Velev and Gao, 2009] investigated 416 ways to encode Hamiltonian cycles to SAT as permutation problems to determine which encodings were the most effective. Further, such a system can take advantage of many built-in common structures in a CAS (e.g., graph families such as hypercubes), which can greatly simplify specifying structures and complex predicates. On the other side, CAS's contain many efficient functions for a broad range of mathematical properties, but often lack the robust search routines available in SAT.

3.1.1 MATHCHECK for Graph Theoretic Problems

The input to MATHCHECK is a tuple $\langle S, \phi \rangle$, where S is a propositional formula extended to allow predicates over graph variables. A graph variable $G = \langle G_V, G_E \rangle$ indicates the vertices and edges that can potentially occur in its instantiation, denoted G_I. A graph variable G consists of a set of $|V|$ Boolean variables (one for each vertex), and $|E|$ Boolean variables for edges. Setting an edge e_{ij} (resp. vertex v_i) to True means that e_{ij} (resp. v_i) is a part of the graph instantiation G_I. Through a slight abuse of notation, we often define a graph variable $G = Q_d$, indicating that the sets of Booleans in G_V and G_E correspond to the vertices and edges in the d-dimensional hypercube Q_d, respectively.

Predicates can be defined by the user, and are classified as either *SAT predicates* or *CAS predicates*. SAT predicates are blasted to propositional logic, using the mapping from graph components (i.e., vertices and edges) to Boolean variables.[1] As an example, for any graph variable G used in an input formula, we add an EdgeImpliesVertices(G) constraint, indicating that an edge cannot exist without its corresponding vertices:

$$\textbf{EdgeImpliesVertices(G):} \bigwedge \{e_{ij} \Rightarrow (v_i \wedge v_j) \mid e_{ij} \in G_E\}. \tag{3.1}$$

CAS predicates, defined as pieces of code in the language of the CAS, check properties of instantiated graphs and add learned clauses to the SAT solver when not satisfied. In our case, we use the SAGE CAS [Stein and Etal., 2010], which we essentially use as a collection of Python modules for mathematics.

Here we provide a high-level overview of the architecture, as depicted in Figure 3.1. Given a formula ϕ over graph variables and predicates, we conjoin the assertions with the negated queries. The **Preprocessor** prepares ϕ for the inner CAS-DPLL loop using standard techniques. First, we create necessary Boolean variables that correspond to graph components (vertices and edges) as described above. We replace each SAT predicate via bit-blasting with its propositional representation in situ (with respect to ϕ's overall propositional structure), such that any assignment found by the SAT solver can be encoded into graphs adhering to the SAT predicates. Finally, the Tseitin-encoding and a Boolean abstraction of ϕ are computed such that CAS predicates are abstracted away by new Boolean variables; since these techniques are well-known, we do not discuss them further. This phase produces three main outputs: the CNF Boolean abstraction ϕ_B of the SAT predicates, a mapping from graph components to Booleans G2B, and a mapping T2B from CAS predicate definitions to Boolean variables. The CAS predicates themselves are fed into the CAS.

[1] For notational convenience, we often use existential quantifiers when defining constraints; these are unrolled in the implementation. We only deal with finite graphs.

The **SAT+CAS** interface acts similar to the DPLL(T) interface between the DPLL loop and theory-plugins, ensuring that partial assignments from the SAT solver satisfy theory-specific CAS predicates. After an assignment is found, literals corresponding to abstracted CAS predicates are checked with respect to the assignment. The SAT+CAS interface provides an API that allows CAS predicates to interact with the SAT solver, which modifies the API from the programmatic SAT solver LYNX [Ganesh et al., 2012]. The potential solution is either deemed a counterexample to the conjecture and returned to the user, or the SAT search is refined with learned clauses. We discuss concrete examples of learned clauses in Section 3.2. Output is either SAT and a counterexample to the conjecture, or UNSAT along with a proof certificate.

Although similar to the DPLL(T) approach of SMT solvers in many aspects, we note several important differences in terms extensibility, power, and flexibility: 1) rather than a monolithic theory plugin for graphs, we opt for a more *extensible* approach by incorporating the CAS, allowing new predicates (say, over numbers, geometry, algebra, etc.) to be easily defined via the CAS functionality; 2) the CAS predicates are defined as pieces of code interpreted by the CAS. This gives considerable *additional power* to the SAT+CAS combination; 3) the user may *flexibly* decide that certain predicates may be encoded directly to Boolean logic via bit-blasting, and thus take advantage of the efficiency of CDCL solvers in certain cases.

3.1.2 Implementation

We have prototyped our system adopting the lazy-SMT solver approach (as in [Sebastiani, 2007]), specifically combining the GLUCOSE SAT solver [Audemard and Simon, 2009] with the SAGE CAS [Stein and Etal., 2010]. Minor modifications to GLUCOSE were made to call out to SAGE whenever an assignment was found (of the Boolean abstraction). The SAT+CAS interface extends the existing SAT interface in SAGE. When the solver determines that the formula is UNSAT, we return two types of certificates. The first is a clausal proof of the final unsatisfiable call to the SAT solver in the DRUP-TRIM format [Heule et al., 2013], which can be checked using the DRUP-TRIM tool. Note however that, unlike a pure SAT solving run, many clauses were added due to checks of CAS predicates. In order to check the correctness of the added clauses, we return a second proof certificate, which consists of the mapping from graph components to Boolean variables, the mapping of abstracted CAS predicates to Boolean variables, and the set of clauses learned from CAS predicate invocations. We discuss how we utilize these certificates in greater detail in Section 3.4.

(a) (b)

Figure 3.2: (a) The red edges denote a generated matching, where the blue vertex 000 is restricted to be unmatched, as discussed in Section 3.2. A Hamiltonian cycle that includes the matching is indicated by the arrows. (b) An edge-antipodal 2-edge-coloring of the cube Q_3. Not a counterexample to Conjecture 2 due to the red (or blue) path from 000 to 111.

3.2 Two Results regarding Open Conjectures of Hypercubes

We use our system to prove two long-standing open conjectures up to a certain parameter (dimension) related to hypercubes, which have not been previously shown. Hypercubes have been studied for theoretical interest, partly due to their properties such as regularity and symmetry, but also for practical uses, such as in networks and parallel systems [Chen and Li, 2010].

3.2.1 Matchings Extend to Hamiltonian Cycles

The first conjecture we look at was posed by Ruskey and Savage on matchings of hypercubes in 1993 [Ruskey and Savage, 1993]; although it has inspired multiple partial results [Fink, 2007, Gregor, 2009] and extensions [Fink, 2009], the general statement remains open:

Conjecture 1 (Ruskey and Savage, [Ruskey and Savage, 1993])**.** For every dimension d, any matching of the hypercube Q_d can be extended to a Hamiltonian cycle.

Consider Figure 3.2a. Recall that a matching is a set of edges that share no vertices. The red edges correspond to a matching and the arrows depict a Hamiltonian cycle extending the matching. Intuitively, the conjecture states that for any d-dimensional hypercube Q_d, no matter which matching M we choose, we can find a Hamiltonian cycle of Q_d that goes through M. Our

encoding searches for matchings, and checks a sufficient subset of the full set of matchings of Q_d to ensure that the conjecture holds for a given dimension (by returning UNSAT and a proof). As we will show, constraints such as ensuring that a potential model is a matching are easily encoded with SAT predicates, while constraints such as "extending to a Hamiltonian cycle" are expressed easily as CAS predicates.

Previous results have shown this conjecture true for $d \leq 4$,[2] however the combinatorial explosion of matchings on higher dimensional hypercubes makes analysis increasingly challenging, and a general proof has been evasive. We demonstrate using our approach the first result that Conjecture 1 holds for Q_5 – the 5-dimensional hypercube. We use a conjunction of SAT predicates to generate a sufficient set of matchings of the hypercube, which are further verified by a CAS predicate to check if the matching can **not** be extended to a Hamiltonian cycle (such that a satisfying model would counterexample the conjecture).

Note that the simple approach of generating *all* matching of Q_d does not scale (see Table 3.1 below), and the approach would take too long, even for $d = 5$. We prove several lemmas to reduce the number of matchings analyzed. In the following, we use the graph variable $G = Q_d$, such that its vertex and edge variables correspond to the vertices and edges in Q_d.

It is straightforward to encode matching constraints as a SAT predicate. For every pair of incident edges e_1, e_2, we ensure that only one can be in the matching (i.e., at most one of the two Booleans may be True), which can be encoded as ("incident?" is a simple predicate that is True if two edges share a vertex):

$$\textbf{Matching(G):} \bigwedge \{(\neg e_1 \vee \neg e_2) \mid e_1, e_2 \in G_E \wedge \text{incident?}(e_1, e_2)\}. \tag{3.2}$$

The number of clauses generated by the above translation is $2^d \cdot \binom{d}{2}$, which can be understood as: for each of the 2^d vertices in Q_d, ensure that each of the d incident edges to that vertex are pairwise not both in the matching.

A previous result from Fink [Fink, 2007] demonstrated that any perfect matching of the hypercube for $d \geq 2$ can be extended to a Hamiltonian cycle. Our search for a counterexample to Conjecture 1 should therefore only consider imperfect matchings, and even further, only maximal forbidden matchings as shown below. First, to encode that we obtain a forbidden matching, we ensure that at least one vertex is not matched by any generated matching. Since all vertices are symmetric in a hypercube, we can, without loss of generality, choose a single vertex v_0 that we ensure is not matched. We encode that all edges incident to v_0 cannot be in the matching:

$$\textbf{Forbidden(G):} \bigwedge \{\neg e \mid e \in G_E \wedge \text{incident?}(v_0, e)\}. \tag{3.3}$$

[2] We were unable to find the original source of the results for $d \leq 4$, however the result is asserted in [Fink, 2007]. We also verified these results using our system.

A further key observation to reduce the matchings search space is that, if a matching M extends to a Hamiltonian cycle, then any matching M' such that $M' \subseteq M$ can also be extended to a Hamiltonian cycle.

Observation 1. All matchings can be extended to a Hamiltonian cycle if and only if all maximal forbidden matchings can be extended to a Hamiltonian cycle.

Proof. The forward direction is straightforward. For the reverse, suppose all maximal forbidden matchings can be extended to a Hamiltonian cycle. For any non-maximal matching M, we can always greedily add edges to M to make it maximal. Call the maximal matching M'. If M' is perfect, Fink's result on perfect matchings can be applied. If not, then it is a maximal forbidden matching, and by assumption it can be extended to a Hamiltonian cycle. In either case, the resulting Hamiltonian cycle must pass through the original matching M. □

We encode this by adding the following constraints to MATHCHECK:

$$\textbf{EdgeOn(G):} \bigwedge \left\{ v \Rightarrow \bigvee \{ e \mid e \in G_E \wedge \text{incident?}(v,e) \} \mid v \in G_V \right\} \tag{3.4}$$

$$\textbf{Maximal(G):} \bigwedge \{ (v_i \vee v_j) \mid e_{ij} \in G_E \}. \tag{3.5}$$

Equation 3.4 states that if a vertex is on, then one of its incident edges must be in the matching. Equation 3.5 ensures that we only generate maximal matchings.

Proposition 1. The conjunction of Constraints 3.1 – 3.5 encode exactly the set of maximal forbidden matchings of the hypercube in which a designated vertex v_0 is prevented from being matched.

Proof. It is clear from above that any model generated will be a forbidden matching by Constraints 3.2 and 3.3 – we prove that Equations 3.4 and 3.5 ensure maximality. Suppose M is a non-maximal matching. Then there exists an edge e such that the matching does not match either of its endpoints. By Constraints 3.1 and 3.4, no edge is incident with either endpoint. But then edge e could be added without violating the matching constraints, and Constraint 3.5 is violated. Thus, any matching generated must be maximal. It remains to show that *all* forbidden maximal matchings that exclude v_0 can be generated. Let M be a forbidden maximal matching such that v_0 is unmatched. We construct a satisfying variable assignment over Constraints 3.1 – 3.5 which encodes M as follows:

$$\begin{aligned} &\{ e \mid e \in M \} \cup \{ \neg e \mid e \in G_E \backslash M \} \cup \\ &\{ v \mid \exists_{e \in M} \text{incident?}(v,e) \} \cup \{ \neg v \mid \nexists_{e \in M} \text{incident?}(v,e) \}. \end{aligned} \tag{3.6}$$

```
1:  ExtendsToHamiltonian()
2:      g ← s.getGraph(G)
3:      q ← CubeGraph(5)
4:      for e in q.edges() do
5:          if e in g
6:              q.setEdgeLabel(e, 1)
7:          else
8:              q.setEdgeLabel(e, 2)
9:          ⟨cycle, weight⟩ ← TSP(q)
10:         return weight == 2 · q.order() − |g|
```

```
1:  AntipodalMonochromatic()
2:      g ← s.getGraph(G)
3:      q ← CubeGraph(6)
4:      pairs ← getAntipodalPairs(q)
5:      for ⟨v1, v2⟩ in pairs do
6:          if shortestPath(g, v1, v2) ≠ ∅
7:              return True      ▷ a path exists
8:      return False
```

Figure 3.3: CAS-defined predicates from each graph theoretic case study. In EXTENDSTOHAMILTONIAN, g corresponds to the matching found by the SAT solver. In ANTIPODALMONOCHROMATIC, g refers to the graph induced by a single color in the 2-edge-coloring.

Constraint 3.2 holds since M is a matching, and therefore no two incident edges can both be in M. Constraint 3.3 holds since it is assumed that v_0 is not matched, and therefore no edge incident to v_0 can be in M. Constraints 3.1 and 3.4 hold simply because they encode the definition of a matched vertex, and the second line of Equation 3.6 ensures that only matched vertices are in the satisfying assignment. Constraint 3.5 holds since M is maximal. □

To check if each matching extends to a Hamiltonian cycle, we create the CAS predicate EXTENDSTOHAMILTONIAN (see Figure 3.3), which reduces the formula to an instance of the traveling salesman problem (TSP). Let M be a matching of Q_d. We create a TSP instance $\langle Q_d, W \rangle$, where Q_d is our hypercube, and W are the edge weights, such that edges in the matching (red edges in Figure 3.2a) have weight 1, and otherwise weight 2 (black edges).

Proposition 2. Let $|V|$ be the number of vertices in Q_d. A Hamiltonian cycle exists through M in Q_d if and only if $\text{TSP}(\langle Q_d, W \rangle) = 2|V| - |M|$.

Proof. Since Q_d has $|V|$ vertices, any Hamiltonian cycle must contain $|V|$ edges. ($\Leftarrow$) From our encoding, it is clear that $2|V| - |M|$ is the minimum weight that could possibly be outputted by TSP, and this can only be achieved by including all edges in the matching and $|V| - |M|$ edges not in the matching. ($\Rightarrow$) The Hamiltonian cycle through M has $|M|$ edges contributing a weight of 1, and $|V| - |M|$ edges contributing a weight of 2. The total weight is therefore $|M| + 2(|V| - |M|) = 2|V| - |M|$. From above, this is also the minimum weight cycle that TSP could produce. □

Finally, after each check of EXTENDSTOHAMILTONIAN that evaluates to True, we add a learned clause, based on computations performed in the predicate, to prune the search space. Since a TSP instance is solved we obtain a Hamiltonian cycle C of the cube. Clearly, any future matchings that are subsets of C can be extended to a Hamiltonian cycle; our learned constraint prevents these subsets (below h refers to the Boolean variable abstracting the CAS predicate):

$$\bigvee\{e \mid e \in Q_{dE} \backslash C\} \cup \{h\}, \text{ where } C \text{ is the learned Hamiltonian cycle.} \tag{3.7}$$

Our full formula for Conjecture 1 is therefore:

$$\begin{aligned} \textbf{assert } & \text{EdgeImpliesVertices}(G) \wedge \text{Matching}(G) \wedge \\ & \text{Forbidden}(G) \wedge \text{EdgeOn}(G) \wedge \text{Maximal}(G) \\ \textbf{query } & \text{ExtendsToHamiltonian}(G) \end{aligned} \tag{3.8}$$

3.2.2 Connected Antipodal Vertices in Edge-antipodal Colorings

The second conjecture deals with edge-antipodal colorings of the hypercube:

Conjecture 2 ([Norine, 2008])**.** For every dimension d, in every edge-antipodal 2-edge-coloring of Q_d, there exists a monochromatic path between two antipodal vertices.

Consider the 2-edge-coloring of the cube in Figure 3.2b. Although the coloring is edge-antipodal, it is not a counterexample, since there is a monochromatic (red) path from 000 to 111, namely $\langle 000, 100, 110, 111 \rangle$. In this case, constraints such as edge-antipodal-ness are expressed with SAT predicates. We ensure that no monochromatic path exists between two antipodal vertices with a CAS predicate. Previous work has shown that the conjecture holds up to dimension 5 [Feder and Subi, 2013] – we show that the conjecture holds up to dimension 6.

We begin with a graph variable $G = Q_6$, and constrain it such that its instantiation corresponds to a 2-edge-coloring of the hypercube. More specifically, since there are only two colors, we associate edges in G's instantiation G_I (i.e., edges evaluated to True) with the color red, and the edges in $Q_d \backslash G_I$ with blue. An important known result is that for a given coloring, the graph induced by edges of one color is isomorphic to the other. It is therefore sufficient to check only one of the color-induced graphs for a monochromatic antipodal path.

We first ensure that any coloring generated is edge-antipodal.

$$\begin{aligned} \textbf{EdgeAntipodal(G):} \bigwedge\{ & (\neg e_1 \wedge e_2) \vee (e_1 \wedge \neg e_2) \\ & \mid e_1, e_2 \in G_E \wedge \text{isAntipodal?}(e_1, e_2)\}. \end{aligned} \tag{3.9}$$

Note that for every edge there is exactly one unique antipodal edge to it. Since there are $2^{d-1} \cdot d$ edges in Q_d, and therefore $2^{d-2} \cdot d$ pairs of antipodal edges, there are $2^{2^{d-2} \cdot d}$ possible 2-edge-colorings that are antipodal. We can reduce the search space by using a recent result from Feder and Suber [Feder and Subi, 2013]:

Theorem 1 ([Feder and Subi, 2013])**.** Call a labeling of Q_d *simple* if there is no square $\langle x, y, z, t \rangle$ such that e_{xy} and e_{zt} are one color, and e_{yz} and e_{tx} are the other. Every simple coloring has a pair of antipodal vertices joined by a monochromatic path.

We therefore prevent simple colorings by ensuring that such a square exists:

$$\begin{aligned} \textbf{NonSimple(G):} \bigvee \{ & (\neg e_{xy} \wedge e_{yz} \wedge \neg e_{zt} \wedge e_{tx}) \vee (e_{xy} \wedge \neg e_{yz} \wedge e_{zt} \wedge \neg e_{tx}) \\ & \mid e_{xy}, e_{yz}, e_{zt}, e_{tx} \in G_E \wedge \text{isSquare?}(e_{xy}, e_{yz}, e_{zt}, e_{tx}) \}. \end{aligned} \tag{3.10}$$

It remains to check whether an antipodal monochromatic path exists, which is checked by the CAS predicate AntipodalMonochromatic in Figure 3.3. Given a graph G, which contains only the red colored edges, we first compute the pairs of antipodal vertices in Q_d. Using the built-in shortest path algorithm of the CAS, we check whether or not any of the pairs are connected, indicating that an antipodal monochromatic path exists. In the case when the predicate returns True, we learn the constraint that all future colorings should not include the found antipodal path P (m abstracts the CAS predicate):

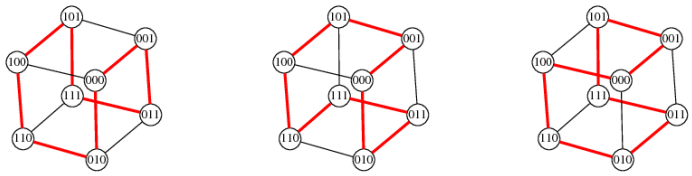
$$\bigvee \{\neg e \mid e \in P\} \cup \{m\}, \text{ where } P \text{ is the learned path.} \tag{3.11}$$

The full formula for Conjecture 2 is then:

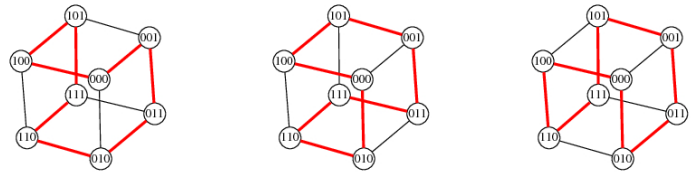
$$\begin{aligned} &\textbf{assert } \text{EdgeImpliesVertices}(G) \wedge \text{EdgeAntipodal}(G) \wedge \text{NonSimple}(G) \\ &\textbf{query } \text{AntipodalMonochromatic}(G) \end{aligned} \tag{3.12}$$

3.2.3 Symmetry Breaking

In each case study, a learned clause is added to the solver whenever the respective CAS predicate is not satisfied by the current model. While the learned clauses described above prune many non-satisfying models from being returned by the solver (e.g., any matching that is a subset of the Hamiltonian cycle in the first case study), many related learned clauses can be obtained through symmetry breaking techniques, due to the highly symmetric nature of hypercubes. Our approach to symmetry breaking is loosely inspired by the work of Benhamou et. al [Benhamou et al., 2010], which proposes an enhanced version of clause learning where all symmetric clauses are learned during conflict analysis, rather than a single conflict clause.



(a) Learned Cycle (b) (1,2)(5,6) (c) (2,4)(3,5)



(d) (1,4,2)(3,5,6) (e) (1,2,4)(3,6,5) (f) (1,4)(3,6)

Figure 3.4: The six unique Hamiltonian cycles of Q_3. The cycle in part (a) is the initially learned cycle, and all are others are derived from (a) using the permutation written in cyclic notation.

Consider Figure 3.4. From the first case study, if we discover the Hamiltonian cycle in Figure 3.4a, then we learn a clause preventing any model that corresponds to a subset of the cycle. Informally, if we fix the vertices of the cube but rotate the Hamiltonian cycle to different orientations, we can learn clauses for each found cycle. Similarly, in the Antipodal case study we can learn many antipodal monochromatic paths through such rotations.

In order to compute such clauses, prior to solving, we compute the automorphism group of the hypercube using the CAS. In our case, the SAGE CAS interfaces the BLISS graph automorphism tool [Junttila and Kaski, 2007]. Then, whenever a Hamiltonian cycle C is learned, for every symmetry π in the automorphism group, we compute $C^\pi = \{(u^\pi, v^\pi) \mid (u,v) \in C_E\}$. It can easily be seen that C^π is also a Hamiltonian cycle of the hypercube due to the properties of symmetries, and we only briefly outline the intuition for this. Suppose $C = \langle c_1, c_2, \ldots, c_n, c_1 \rangle$. For any edge in the cycle (c_i, c_{i+1}), we know that (c_i^π, c_{i+1}^π) is an edge in the cube since symmetries preserve the set of edges and non-edges. Finally, since π is a permutation of the vertices, $c_1^\pi, \ldots, c_n^\pi$ are unique, so C^π is a Hamiltonian cycle.

The Antipodal case study is handled analogously. We note that performing this operation over the entire automorphism group can generate many redundant clauses; we ensure that duplicates are not added. This can be optimized by considering only the proper symmetry group, which

Dimensions	Matchings	Forbidden Matchings	Maximal Forbidden Matchings
2	7	3	0
3	108	42	2
4	41,025	14,721	240
5	13,803,794,944	4,619,529,024	6,911,604

Table 3.1: The number of each type of matching in the cube. If a problem-specific tool iterated through all matchings, the number in each cell corresponds to the number of cases that would need to be tried. The number of matchings of the hypercube were computed using our tool in conjunction with sharpSAT [Thurley, 2006]: a tool for the #SAT problem. Note that the numbers for forbidden matchings are only lower bounds, since we only ensure that the *origin* vertex is unmatched. However, any unfound matchings are isomorphic to found ones.

omits any symmetries from reflection.

3.3 Performance Analysis of MATHCHECK

For the two graph theoretic conjectures, we ran Formula 3.8 with $d = 5$ and Formula 3.12 with $d = 6$ until completion. Since both runs returned UNSAT, we conclude that both conjectures hold for these dimensions, which improves upon known results for both conjectures.

The experiments were performed on a 2.4 GHz 4-core Lenovo Thinkpad laptop with 8GB of RAM, running 64-bit Linux Mint 17. We used SAGE version 6.3 and GLUCOSE version 3.0. Formula 3.8 required 348,150 checks of the EXTENDSTOHAMILTONIAN predicate, thus learning an equal number of Hamiltonian cycles in the process, and took just under 8 hours. Formula 3.12 required 86,612 checks of the ANTIPODALMONOCHROMATIC predicate (learning the same number of monochromatic paths), requiring 1 hour 35 minutes of runtime. We note that for lower dimensional cubes solving time was far less (< 20 seconds for either case study). Adding symmetry breaking greatly reduced the solving time and number of CAS predicate checks: the first case study required 1 hour and 5 minutes, and 2441 CAS predicate checks, while the second took only 3 minutes and 122 predicate checks.

The approach we have described significantly dominates naïve brute-force approaches for both conjectures; learned clauses greatly reduce the search space and cut the number of necessary CAS predicate checks. Given the data in Table 3.1 and the number of calls to EXTENDSTOHAMILTONIAN for Q_5, a brute-force check of all matchings (resp. forbidden matchings) of Q_5 would require 39,649 (resp. 20) times more checks of the predicate (i.e., that many more TSP calls) than our approach. Similar comparisons can be made for the second case study.

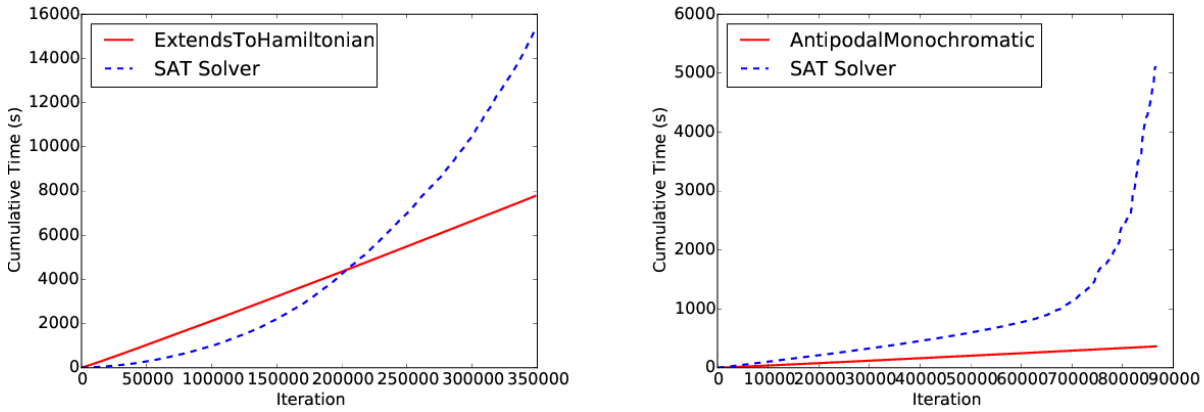


Figure 3.5: Cumulative times spent in the SAT solver and CAS predicates during the two graph theory case studies. SAT solver performance degrades during solving (as indicated by the increasing slope of the line), due to the extra learned clauses and more constrained search space.

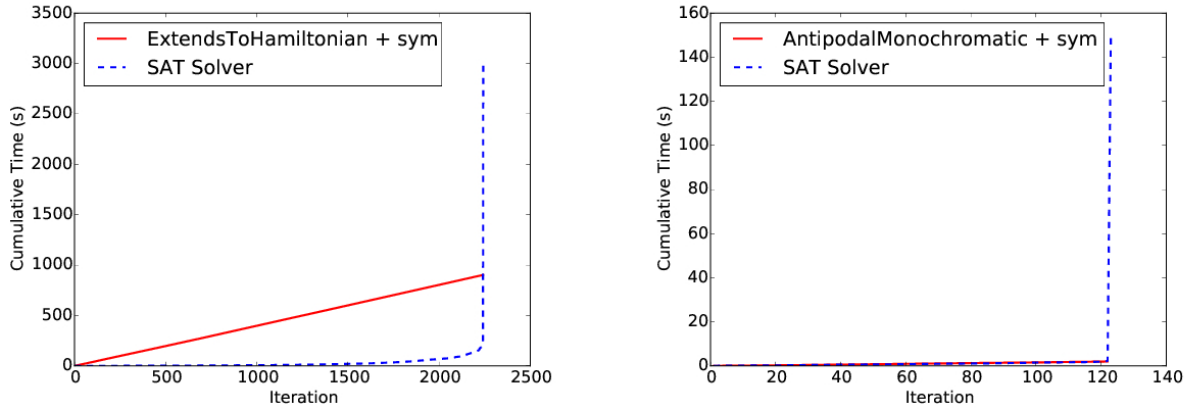


Figure 3.6: Cumulative times spent in the SAT solver and CAS predicates during the two graph theory case studies with symmetry breaking. Note the differing axes from Figure 3.5. Interestingly, solving time is now dominated by the last UNSAT call.

Figure 3.5 depicts how much time is consumed by the SAT solver and CAS predicates in both case studies, and Figure 3.6 indicates the same but with symmetry breaking enabled. The lines denote the cumulative time, such that the right most point of each line is the total time consumed by the respective system component. The near-linear lines for the CAS predicate calls indicate that each check consumed roughly the same amount of time. SAT solving ultimately dominates the runtime in both case studies, particularly due to later calls to the solver when many

learned clauses have been added by CAS predicates, and the search space is highly constrained. Interestingly, the final UNSAT call to the SAT solver when symmetry breaking was enabled required significantly more time than any other calls. We did not experience this behaviour in the non-symmetry breaking experiments.

One of our motivations for this work was to allow complicated predicates to be easily expressed, so it is worth commenting on the size of the actual predicates. Since predicates were written using SAGE (which is built on top of Python), the pseudocode written in Figure 3.3 on page 25 matches almost exactly with the actual code, with small exceptions such as computing the antipodal pairs in the second one. All other function calls correspond to built-in functions of the CAS. "Learn-functions," which were used to generate learned clauses from CAS predicate results, were also short, requiring less than 10 lines of code each.

3.3.1 Analysis of Case Studies with Existing SAT-based Approaches

We were interested in finding previously existing SAT-based tools capable of efficiently solving and expressing the problems in our two graph theoretic case studies. Our criteria for selecting a tool were that both case studies could be succinctly expressed, and solved for at least lower-dimensional cubes with reasonable efficiency. We excluded standard SMT solvers from this evaluation due to poor support for higher-order logic (particularly in terms of performance). Since we are dealing with finite cases, one could in theory compare against "bitblasting" approaches, akin to how Hamiltonian cycle constraints are expressed in SAT solvers [Velev and Gao, 2009]. However, since our formula requires that *no* Hamiltonian cycle exists through the matching, encoding techniques from [Velev and Gao, 2009], which check for the *existence* of a Hamiltonian cycle, cannot be succinctly used to encode our formula.[3] We discuss other related tools in Section 3.5.

One such tool that met these criteria was ALLOY*, a relational finite model finder for higher-order logic [Milicevic et al., 2015], which extends its first-order predecessor [Jackson, 2012]. Alloy (for first-order logic) translates input to the constraint solver KODKOD [Torlak, 2009], which performs an efficient translation to SAT. ALLOY* extends this approach with a CEGAR loop on top of KODKOD; abstraction avoids solving higher-order constraints directly, and refinement

[3] In [Velev and Gao, 2009], the Hamiltonian cycle detection problem is reduced to SAT by encoding it as a permutation problem, such that if a Hamiltonian cycle exists, then the permutation extracted from the model corresponds to the cycle. In order to negate this existential check (as is needed by Formula 3.8), one must ensure that all permutations of the vertices do not correspond to a Hamiltonian cycle. While this can be succinctly expressed a quantified Boolean formula (QBF), a substantial number of universally quantified variables must be introduced. Hamiltonian cycle detection can also be succinctly expressed in answer set programming (ASP) [Lifschitz, 2008], but similar search-space explosion problems still exist.

Case Study	Translation (s)	Solving (s)	#Vars	#Clauses	MATHCHECK Time (s)
Matchings(3)	1.583	3.051	8037	27370	0.160
Matchings(4)	N/A	N/A	N/A	N/A	0.961
Matchings(5)	N/A	N/A	N/A	N/A	28800.000
Antipodal(3)	0.511	0.054	18366	65927	0.463
Antipodal(4)	4.201	0.268	203066	838251	2.211
Antipodal(5)	92.627	4.091	2221834	10682619	28.035
Antipodal(6)	N/A	N/A	N/A	N/A	3900.000

Table 3.2: Translation and solving times for ALLOY* on the two graph theory case studies (hypercube dimension indicated in parentheses). The number of variables and clauses produced by ALLOY*'s translation to SAT are likely the reason for long translation times. Times for MATHCHECK indicate total time; translation times were negligible compared to solving.

ensures that models that do not satisfy the elided higher-order constraints are avoided in future iterations of the CEGAR loop.

For our experiments, we used default options for ALLOY*, however we changed the underlying SAT solver to GLUCOSE (which is generally considered faster than the default solver SAT4J), to match our experiments, and increased maximum memory to 4GB, which was the maximum allowed by their user interface. We also increased the maximum CEGAR loop iterations to 100,000, although this limit was never reached. For encoding Hamiltonicity, we used the monadic second-order logic encoding from [Downey and Fellows, 2013, p. 247], with slight modifications due to ALLOY*'s concise syntax and for performance improvements. For ensuring that a monochromatic path exists between two antipodal vertices, we used a previously encoded connected constraint that is available with the ALLOY* implementation. We also made use of ALLOY*'s `when` construct, which improves performance of the CEGAR loop on quantified implications. All encodings are available at [Zulkoski and Ganesh, 2015].

Table 3.2 displays the time taken by ALLOY* to translate its input to SAT and then perform solving. Recall that solving may require many calls to the SAT solver due to the CEGAR loop. We also included the total number of variables and clauses in the initial translation to SAT. ALLOY* produces an error during the translation phase for the Matching case study for $d = 4$, and the Antipodal case study for $d = 6$, presumably due to memory constraints and the large CNF formulas generated. Interestingly, running time is completely dominated by the translation for the Antipodal case study. The long translation time is due to the large increase in problem size when converting from relational first order logic to SAT for these particular problems. In addition, their approach does not take advantage of predicate-specific learning opportunities, such as preventing any future matchings that are subsets of found cycles in the Matchings case study.

3.4 Verification of Results

Given the mathematical nature of our results, it is important to have a high degree of confidence in their correctness. This is especially true when trying to disprove a statement. One has to rely on the correctness of the encodings, the theory, and the tools that have been used. Two main issues typically arise when verifying SAT-based analysis: 1) one must ensure that input to the SAT solver is correct, i.e., the tool which generates the DIMACS file correctly encodes the problem; 2) the computation of the SAT solver is correct. Other SAT-based analyses of mathematical problems, such as the "SAT attack" on the Erdős-discrepancy problem by Konev et al. [Konev and Lisitsa, 2014] or the work of Heule et al. [Heule et al., 2016a] on solving the Boolean Pythagorean triples problem, mitigate these concerns in primarily two ways. First, in each of these works, the input DIMACS files could be generated by a very small program, which could be checked manually. In both cases, these generators were publicly released in order to be independently validated. Second, SAT solver proofs were verified with off-the-shelf clausal proof verifiers, such as DRUP-TRIM [Heule et al., 2013] or the more recent DRAT-TRIM [Wetzler et al., 2014].

Several issues arise when trying to take similar steps for our results. First, our tool has grown to several thousands of lines of code, and relies on multiple other software systems such as SAGE [Stein and Etal., 2010], GLUCOSE [Audemard and Simon, 2009], and various Python libraries. As such, manually verifying the correctness of such a system is a non-trivial task. In order to strengthen the confidence in our results, we instead provide separate SAT generators for our two graph theory case studies, independent from the rest of our tool's codebase, that are small enough to be manually checked (approximately 100 lines of Python code each). Second, since clauses are periodically added to the solver via external calls to the CAS, merely checking the proof produced by the final UNSAT call to the SAT solver is insufficient. We must additionally ensure that clauses returned by the CAS predicates adhere to their specifications, i.e., Formulas 3.7 and 3.11. We discuss the independent checkers and certificates in more detail in the following sections.

3.4.1 UNSAT Proof Certificates

When MATHCHECK returns UNSAT, two types of proof certificates are produced. The first is a DRUP-TRIM certificate [Heule et al., 2013] from the final unsatisfiable call to the SAT solver. This is then checked with DRUP-TRIM to verify the correctness of the SAT solver's resolution proof; since this approach is commonly used we do not elaborate further. The proof for our Matchings case study was 927 MB in size, and for the Antipodal case study it was 1.4 GB (for the highest dimension cubes checked). In both of our case studies, DRUP-TRIM verified the proofs produced by the SAT solver. We also verified the results for lower dimensional cubes.

The second certificate is used to check the clauses produced by the CAS. The certificate is a triple $\langle M, P, C \rangle$, where M is a bijection between graph components (i.e., vertices and edges) and DIMACS variables, P is a similar mapping from abstracted CAS predicates and their corresponding DIMACS variables, and C is the set of learned clauses produced by the CAS predicates. We additionally annotate which CAS predicate produced each clause. The purpose of this certificate is to verify that the learned clauses produced by the CAS predicates adhere to their specification. This involves creating specialized checkers for each predicate. For example, consider a certificate $\langle M, P, C \rangle$ produced by the Matchings case study. It may be useful to refer to Formula 3.7. For a given learned clause, we first ensure that all literals occur positively, and then lift all DIMACS variables to their associated graph components/CAS predicates (e.g., the abstraction h of EXTENDSTOHAMILTONIAN) using M and P. We ensure that, for example, h exists in the learned clause, and that all remaining variables correspond to edges in the graph (as opposed to vertices). Finally, we check that the set of edges *not* represented in the clause correspond to a Hamiltonian cycle of Q_d. We repeat this process for every learned clause produced during solving.

3.4.2 Correctness of Specification

As discussed, ensuring the correctness of a large system is non-trivial, and testing that the tool correctly encodes the problem to SAT may not be sufficient. For our two graph theory case studies, we opted to create separate DIMACS generators that are much more concise than MATHCHECK's code base (approximately 100 lines of code each). These generators however only directly generate clauses related to the SAT predicates, and rely upon the certificate produced by MATHCHECK (which is also checked) to add the clauses generated by the CAS predicates (e.g., clauses associated with learned Hamiltonian cycles). One additional complication is that since these learned clauses adhere to the mapping between graph components and DIMACS used when MATHCHECK solved the formula, we must use the same mapping when generating DIMACS. We therefore ensure that the graph components used in our generators correctly adhere to the mapping specified in the first field M of the certificate, as discussed previously. Finally, before adding the learned clauses to the DIMACS file, we check that they correspond to Formulas 3.7 and 3.11, using specialized checkers as described previously. In both case studies, the generated SAT formulas were unsatisfiable, as expected. We again verified these results with DRUP-TRIM.

3.4.3 Further Threats to Correctness

While we strive to ensure correctness of as much of our tool as possible, since it has not been formally verified, we do make some assumptions regarding correctness. Specifically, we do not

check that communication between the SAT solver and CAS is correct, in the sense that the mapping between graph components and DIMACS variables remains constant. Second, we base our checks on the assumption that the human-derived proofs from Section 3.2 are correct. Ideally, these proofs would be verified with a theorem prover such as COQ [COQ development team, 2004] or ISABELLE [Nipkow et al., 2002]. Nonetheless, we believe that our current approach gives a high-degree of confidence in the correctness of our results.

3.5 Related Work

Our approach of combining a CAS system within the inner-loop of a SAT solver most closely resembles and is inspired by DPLL(T) [Nieuwenhuis et al., 2004]. There are also similarities with the idea of programmatic SAT solver LYNX [Ganesh et al., 2012], which is an instance-specific version of DPLL(T). Our tool MATHCHECK is inspired by the recent SAT-based results on the Erdős discrepancy conjecture [Konev and Lisitsa, 2014]. Within the constraint satisfaction programming (CSP) domain, lazy clause learning was introduced in [Ohrimenko et al., 2009] to extract clausal reasons from constraints during backtracking search, which was built upon work [Katsirelos and Bacchus, 2005] that generalizes CSP nogoods (akin to clauses in CDCL) to allow both assignments and non-assignments of variables. Other works [Dooms et al., 2005, Gebser et al., 2014, Soh et al., 2014] have extended solvers to handle graph constraints, by either creating solvers for specific graph predicates [Gebser et al., 2014, Soh et al., 2014], or by defining a core set of constraints with which to build complex predicates [Dooms et al., 2005]. Our approach contains positive aspects from both: state-of-the-art algorithms from the CAS can be used to define new predicates easily, and the methodology is general, in that new predicates can be defined using the CAS. A recent solver called MONOSAT is capable of efficiently solving problems involving monotonic theories [Bayless et al., 2015]; in particular it supports many graph properties such as shortest path, connectedness, minimum spanning tree, etc. An efficient encoding for the edge-antipodal colorings conjecture may be possible using their approach, however the Ruskey-Savage conjecture violates the monotonic theory requirement. ALLOY* [Milicevic et al., 2015] is capable of solving many bounded higher-order relational logic specifications, and can therefore support the types of problems addressed in our case studies. We showed in Section 3.3.1 that the encodings of the two graph theory case studies do not seem to scale in ALLOY*, partly due to the time needed to translate the problem to the large SAT formulas generated during solving.

Several tools have combined a CAS with SMT solvers for various purposes, mainly focusing on the non-linear arithmetic algorithms provided by many CAS's. For example, the VERIT

SMT solver [Bouton et al., 2009] also uses functionality of the REDUCE CAS[4] for non-linear arithmetic support. Our work is more in the spirit of DPLL(T), rather than modifying the decision procedure for a single theory.

Symmetry breaking has been a widely studied topic in the context of SAT [Sakallah, 2009, Benhamou et al., 2010, Aloul et al., 2003], constraint solving [Gent et al., 2006, Gent and Smith, 1999, Torlak, 2009], and more recently SMT [Déharbe et al., 2011, Areces et al., 2013]. Symmetry breaking approaches are either static – constraints are added before solving to prevent isomorphic models, as in [Torlak, 2009], or dynamic – symmetries are detected during search and appropriate clauses are added, as in [Gent and Smith, 1999, Benhamou et al., 2010]. Our approach is most inspired by [Benhamou et al., 2010] – rather than learning a single learned clause from an unsatisfied CAS predicate, many are learned that correspond to graphs isomorphic to the one found (e.g., the Hamiltonian cycle).

3.6 Conclusions

In this chapter, we presented MATHCHECK, a combination of a CAS in the inner-loop of a conflict-driven clause-learning SAT solver, and we show that this combination allows for highly expressive predicates that are otherwise non-trivial/infeasible to encode as purely Boolean formulas. Our approach combines the well-known domain-specific abilities of CAS with the search capabilities of SAT solvers thus enabling us to finitely verify two long-standing open mathematical conjectures over hypercubes up to to particular dimension, not feasible by either kind of tool alone. We further discussed how our system greatly dominates naïve brute-force search techniques for the case studies. Known symmetry breaking techniques further drastically reduced solving times. We stress that the approach is not limited to the domain of graph theory, and has since been extended to investigate other domains, such as the study of Hadamard matrices [Bright, 2017].

[4] http://www.reduce-algebra.com/index.htm

Chapter 4

Relating SAT Formula Measures to CDCL Solving

For more general classes of formulas than considered in the previous chapter, it remains unclear which properties of formulas best relate to and explain solver performance. This chapter consists of a broad attempt at relating many of the most popular measures of Boolean formulas to SAT solving, first by investigating which measures correlate best with solving time, and then analyzing which measures are best at distinguishing instances from different categories. While this may be a useful first step, it still says little about *why* one measure may better explain SAT performance than another. We take steps toward addressing this issue by considering how several of the considered measures relate to the behaviour of CDCL solvers during search. Our findings suggest that the measures considered can be a useful *lens* for explaining the locality-based nature of CDCL SAT solvers.

Main Contributions:

- **Correlations between structural measures and CDCL performance.** We performed a large scale evaluation of several structural measures over 7000 SAT competition instances, and show that while many of the considered measures correlate well in the small, i.e., over sub-categories of application instances, the correlations are not strong when considering large sets of diverse instances.
- **Classifying instances with structural measures.** We build upon the experiments in [Ansótegui et al., 2017] by constructing classifiers from various sets of structural measures. We consider larger sets of features and approximately 800 instances from the application track of previous SAT competitions. We further show that application and

agile instances have significantly better structural parameter values compared to random or crafted instances, on average.

- **Analyzing CDCL behaviour vis-à-vis structural properties.** We show how these structural parameters can be used as a lens to analyze the effect of various solving heuristics, with a current focus on branching heuristics and restart policies. We discuss three sets of experiments based upon community structure, backbones, and LSR backdoors.

- **Further analysis available in Appendix A.** For most of the experiments in this Chapter, we provide additional experiments in the appendix. For correlation studies, we consider different solvers for our dependent runtime variable, as well as different regression models. We also fine-grain our analyses by differentiating satisfiable and unsatisfiable instances.

4.1 Experimental Setup

Table 4.1 lists out the considered measures along with a brief description of how they are computed. We use off the shelf tools to compute weak backdoors [Li and Van Beek, 2011], community structure and modularity [Newsham et al., 2014], backbones [Janota et al., 2015], variable popularity [Ansótegui et al., 2009], fractal dimension [Ansótegui et al., 2014], and treewidth [Mateescu, 2011]. Their approaches are briefly described in Table 4.1. Due to the difficulty of exactly computing these parameters, with the exception of backbones, the algorithms used in previous work (and our experiments) do not find optimal solutions, e.g., the output may be an upper-bound on the size of the minimum backdoor.

In Chapters 5 and 6, we introduce new measures which we briefly describe here. We compute LSR backdoors using a tool we developed called *LaSeR*, which computes an upper-bound on the size of the minimal LSR backdoor. The tool is built on top of the MapleSAT SAT solver [Liang et al., 2016b], an extension of MiniSat [Eén and Sörensson, 2003]. We further describe the *LaSeR* algorithm in Section 6.5. Resolvability and mergeability, as defined in Section 2.3.3, are computed by simply iterating over all pairs of clauses and checking how many resolve or merge (counting a pair n times if there are n merge literals).

Table 4.2 shows the data sources for our experiments. We include all instances from the application and crafted tracks of the SAT competitions from 2009 to 2014, as well as the 2016 agile track. We additionally included a small set of random instances as a baseline. As the random instances from recent SAT competitions are too difficult for CDCL solvers, we include a set of instances from older competitions. We pre-selected all random instances from the 2007 and 2009 SAT competitions that could be solved in under 5 minutes by MapleCOMSPS

Type	Benchmarks	Unsat?	Tool Description
Weak Backdoors [Kilby et al., 2005, Li and Van Beek, 2011, Williams et al., 2003b]	3SAT, GC, SR	No	Perform Tabu-based local search to minimize the number of decisions in the final model.
LS Backdoors [Dilkina et al., 2009b, Dilkina et al., 2014]	LP	Yes	Run a clause-learning solver, recording all decisions, which constitutes a backdoor.
Backbones [Janota et al., 2015, Kilby et al., 2005, Williams et al., 2003b]	3SAT, GC, Comps	No	Repeated SAT calls with UNSAT-core based optimizations.
Treewidth [Liang et al., 2015a, Mateescu, 2011]	C09, FM	Yes	Heuristically compute residual graph G. The max-clique of G is an upper-bound.
Modularity [Ansótegui et al., 2012, Newsham et al., 2014]	Comps	Yes	The *Louvain method* [Blondel et al., 2008] – greedily join communities to improve modularity.
Fractal Dimension [Ansótegui et al., 2014]	Comps	Yes	Heuristically cover vertices of the graph with circles of diameter l. Compute the rate at which the number of needed circles to cover the graph decreases as l increases.
Variable Popularity α_V [Ansótegui et al., 2009]	Comps, 3SAT	Yes	Estimate the exponent of the power-law distribution of variable occurrences.
Resolvability + Mergeability [Chapter 5]	–	Yes	Iterate through all pairs of clauses counting the number of resolvable/mergeable pairs.
LSR Backdoors [Chapter 6]	–	Yes	Run a clause-learning solver, recording all variables that occur in clauses of the proof.

Table 4.1: Previously studied benchmarks for each considered parameter, as well as description of tools used to compute them. The "Unsat?" column indicates if the parameter is defined on unsatisfiable instances. Abbrevations: 3SAT – random 3-SAT; GC – graph coloring; LP – logistics planning; SR – SAT Race 2008; C09 – SAT competition 2009; Comps – 2009-2014 SAT competitions; FM – feature models.

	Instances	LSR	Weak	Cmty	Bones	TW	α_V + Dim	Merge+Res
Agile	4968	3592	464	4968	208	4968	4901	4870
Application	1144	702	281	891	193	1088	1144	824
Crafted	753	428	195	613	154	753	753	604
Random	126	125	76	126	59	126	126	126
Total	6991	4847	1016	6598	614	6935	6924	6424

Table 4.2: Depicts the number of instances in each benchmark, as well as the number of instances for which we were able to successfully compute each metric/time.

[Liang et al., 2016c], the winner of the main track of the 2016 SAT competition. All instances were simplified using MapleCOMSPS' preprocessor before computing the parameters. The preprocessing time was not included in solving time.

In the 2013 SAT competition, each of the 300 instances in the application category were classified into 19 different sub-categories: *2d-strip-packing, bio, crypto-aes, crypto-des, crypto-gos, crypto-md5, crypto-sha, crypto-vmpc, diagnosis, hardware-bmc, hardware-bmc-ibm, hardware-cec, hardware-velev, planning, scheduling, scheduling-pesp, software-bit-verif, software-bmc* and *termination*. For certain experiments, we cluster instances based upon sub-category. We include all instances from the application track of the SAT competitions from 2009-2014 that we could accurately place in the 19 sub-categories for these experiments,[1] in total 795 instances.

Experiments were run on an Azure cluster, where each node contained two 3.1 GHz processors and 14 GB of RAM. Several longer running experiments were run on the SHARCNET Orca cluster [SHARCNET, 2017], where nodes contain cores between 2.2 GHz and 2.7 GHz. Each experiment was limited to 6 GB of RAM. For the application, crafted, and random instances, we allotted 5000 seconds for solving (the same as used in the SAT competition), 24 hours for backbone and weak backdoor computation, 3 hours for LSR backdoor computation, and 2 hours for all other computations. For the agile instances, we allowed 60 seconds for MapleCOMSPS solving and 300 seconds for LSR backdoor computation; the remaining agile parameter computations had the same cutoff as application. Due to the difficulty of computing these parameters, we do not obtain values for all instances due to time or memory limits being exceeded. The number of data points obtained for each measure and benchmark is given in Table 4.2.

[1]The only SAT competition with publicly available labels of the instances into sub-categories is from 2013. However, the naming conventions between SAT competitions is fairly standardized, making it easy to manually classify most instances. Approximately 20% of the instances could not be obviously classified, and were not included.

4.2 Correlating Structural Measures with Solving Time

Our first set of experiments investigate the relationship between structural parameters and CDCL performance. Specifically, we pose the following question: *Do parameter values correlate with solving time? In particular, can we build significantly stronger regression models by incorporating combinations of these features?*

To address this, we construct both linear regression and ridge regression models from subsets of features related to these parameters. Although previous work has considered primarily linear regression models (e.g. [Newsham et al., 2014]), in our experiments, we observed that high multi-collinearities existed in the features, which may cause inaccuracy in the regression results. Ridge regression is used to address the issue of collinear data. Nonetheless, we report results for both models for completeness.

We consider the following "base" features: number of variables (V), number of clauses (C), number of communities (Cmtys), modularity (Q), weak backdoor size (Weak), the number of minimal weak backdoors computed (#Min_Weak), LSR backdoor size (LSR), treewidth upper-bound (TW), fractal dimension of the VIG and CVIG (Dim_V and Dim_C, respectively), variable popularity power-law exponent (α_V), and backbone size (Bones). For each $P \in \{C, Cmtys, Weak, LSR, TW, Bones\}$ we include its ratio with respect to V as P/V. We also include the ratio feature $Q/Cmtys$, as used in [Newsham et al., 2014], as well as the ratio of mergeability over resolvability: M/R.

There are clearly many more features that could be chosen for our study; we considered this particular set of features for several reasons. First, most of the considered features have already been evaluated on smaller classes of instances (either with mixed or somewhat positive results), as in Table 4.1. The second reason is more pragmatic: certain other interesting features are simply too hard to compute over the large set of instances we considered in this study. For example, we know of no efficient algorithm to compute, or even approximate, non-trivial strong backdoors of SAT formulas at this time.

All features are normalized to have mean 0 and standard deviation 1. For a given subset of these features under consideration, we use the "$\oplus$" symbol to indicate that our regression model contains these base features, as well as all higher-order combinations of them (combined by multiplication). For example, $V \oplus C$ contains four features: V, C, and $V \cdot C$, plus one "intercept" feature. Our dependent variable is the log of runtime of the MapleCOMSPS solver. Our results using linear regression are displayed in Table 4.3, and the results using ridge regression are in Table 4.4.

In each of the tables, we first consider sets of features defined with respect to a single parameter type, e.g., only weak backdoor features, or only community structure based features, along with V

Feature Set	Application	Crafted	Random	Agile
$V \oplus C \oplus C/V$	0.03 (1143)	0.04 (753)	0.08 (126)	0.85 (4968)
$V \oplus C \oplus Cmtys \oplus Q \oplus Q/Cmtys$	0.07 (889)	0.26 (613)	0.28 (126)	0.89 (4968)
$V \oplus C \oplus LSR \oplus LSR/V$	0.27 (702)	0.46 (428)	0.41 (125)	0.91 (3592)
$V \oplus C \oplus \#Min\_Weak \oplus Weak$	0.14 (274)	0.15 (195)	0.23 (76)	0.56 (464)
$V \oplus C \oplus Bones \oplus Bones/V$	0.17 (193)	0.40 (154)	0.04 (59)	0.43 (208)
$V \oplus C \oplus TW \oplus TW/V$	0.05 (1087)	0.07 (753)	0.28 (126)	0.91 (4968)
$V \oplus C \oplus Dim_V \oplus Dim_C \oplus \alpha_V$	0.06 (1143)	0.19 (753)	0.25 (126)	0.88 (4901)
$V \oplus C \oplus M \oplus R \oplus M/R$	0.20 (823)	0.25 (604)	0.19 (126)	0.91 (4870)
$C/V \oplus Q \oplus TW/V \oplus M/R \oplus R \oplus Dim_V$	**0.31 (823)**	0.43 (604)	0.15 (126)	0.95 (4870)
$Q \oplus Q/Cmtys \oplus TW/V \oplus M/R \oplus Dim_V \oplus Dim_C$	0.30 (823)	**0.56 (604)**	0.39 (126)	0.95 (4870)
$V \oplus Cmtys \oplus Q \oplus R \oplus Dim_V \oplus \alpha_V$	0.22 (823)	0.42 (604)	**0.66 (126)**	0.93 (4870)
$V \oplus C/V \oplus Q \oplus TW/V \oplus M \oplus Dim_C$	0.23 (823)	0.47 (604)	0.20 (126)	**0.96 (4870)**

Table 4.3: Adjusted R^2 values for the given features using linear regression, compared to log of MapleCOMSPS' solving time. The number in parentheses indicates the number of instances that were considered in each case. The lower section considers heterogeneous sets of features across different parameter types.

Feature Set	Application	Crafted	Random	Agile
$V \oplus C \oplus C/V$	0.01 (1143)	0.03 (753)	0.06 (126)	0.75 (4968)
$V \oplus C \oplus Cmtys \oplus Q \oplus Q/Cmtys$	0.05 (889)	0.11 (613)	0.13 (126)	0.79 (4968)
$V \oplus C \oplus LSR \oplus LSR/V$	0.14 (702)	0.30 (428)	0.18 (125)	0.81 (3592)
$V \oplus C \oplus \#Min\_Weak \oplus Weak$	0.02 (274)	0.12 (195)	0.06 (76)	0.37 (464)
$V \oplus C \oplus Bones \oplus Bones/V$	0.17 (193)	0.29 (154)	0.08 (59)	0.14 (208)
$V \oplus C \oplus TW \oplus TW/V$	0.03 (1087)	0.05 (753)	0.06 (126)	0.87 (4968)
$V \oplus C \oplus Dim_V \oplus Dim_C \oplus \alpha_V$	0.02 (1143)	0.07 (753)	0.11 (126)	0.79 (4901)
$V \oplus C \oplus M \oplus R \oplus M/R$	0.08 (823)	0.11 (604)	0.08 (126)	0.73 (4870)
$C \oplus Cmtys \oplus TW/V \oplus R \oplus M/R \oplus Dim_V$	**0.15 (823)**	0.17 (604)	0.11 (126)	0.88 (4870)
$C \oplus Q \oplus Q/Cmtys \oplus M/R \oplus Dim_V \oplus \alpha_V$	0.11 (823)	**0.27 (604)**	0.13 (126)	0.79 (4870)
$V \oplus C \oplus Q \oplus Q/Cmtys \oplus TW/V \oplus Dim_V$	0.09 (823)	0.15 (604)	**0.19 (126)**	0.89 (4870)
$V \oplus C \oplus C/V \oplus Cmtys \oplus Q \oplus TW/V$	0.08 (823)	0.12 (604)	0.11 (126)	**0.90 (4870)**

Table 4.4: Repeated results using ridge regression.

and C as baseline features. Each cell reports the adjusted R^2 value of the regression, as well as the number of instances considered in each case (which corresponds to the number of instances for which we have data for each feature in the regression). It is important to note that since different subsets of SAT formulas are used for each regression (since our dataset is incomplete), we should not directly compare the cells in the top section of the table. Nonetheless, the results do give some indication if each parameter relates to solving time.

In order to show that combinations of these features can produce stronger regression models, in the bottom half of Tables 4.3 and 4.4, we consider all instances for which we have all data collected, excluding LSR backdoors, weak backdoors, and backbones. We exclude backbones and weak backdoors in this case, as it limits our study to satisfiable instances and greatly reduces the number of datapoints. LSR backdoors further limit the number of instances and are not included. We considered all subsets of base features of size 6 (e.g. $C/V \oplus Q \oplus TW/V \oplus M/R \oplus R \oplus Dim_V$), and report the best model for each benchmark, according to adjusted R^2 (i.e. the bolded data along the diagonal). The feature coefficients for each of these best models using linear regression and MapleCOMSPS as the solver are listed in Appendix A. This results in somewhat stronger correlations than with any of the features sets defined over a single parameter (i.e. the top half of the table). Although we report our results with six base features (whereas the top half of the table typically only used four), similar results appear if we only use four base features. We note that R^2 values results tend to be significantly higher for the Agile instances, as compared to application and crafted instances. This is somewhat expected, as the set of instances are all derived from the SAGE whitebox fuzzer [Godefroid et al., 2008], as compared to our other benchmarks which come from a heterogeneous set of sources.

We further considered how each measure correlated with solving time within sub-categories of the application instances. Table 4.5 shows the Pearson (above) and Spearman (below) correlations when using MapleCOMSPS as the solver. We highlight correlation values of $|r| \geq 0.4$; empty cells in the table denote worse correlations. Full numerical data is available in Appendix A.

4.2.1 Interpretation of Results

We first note that, not too surprisingly, no single parameter is significantly predictive of SAT solver performance across the wide variety of instances considered in this work. While combinations of these parameters do produce notable improvements with respect to R^2 values, there is still much room for improvement, especially when considering the application instances. Even though the parameters we used in our study are arguably some of the more "popular" ones discussed in SAT solver literature, when used to examine the types of instances that SAT solvers often tackle in practice, there seems to be little correlation with performance. We hope that this work will inspire

Pearson	C/V	Q	Bones/V	TW/V	Weak/V	LSR/V	M	M/R
2d-strip-packing	++	+		–		++		
argumentation		– –		+	+++	+++	+++	+
bio	+		+++	+				
crypto-aes			–		+			
crypto-des				– –		+		+
crypto-gos	+	+		–			+	+
crypto-md5								
crypto-sha			+					
crypto-vmpc	+		–	+			–	–
diagnosis								
hardware-bmc						+	– –	
hardware-bmc-ibm				+			+++	+++
hardware-cec								
hardware-manolios				–				– –
hardware-velev						– – –		–
planning		–						
scheduling								
scheduling-pesp								–
software-bit-verif								
termination						++		

Spearman	C/V	Q	Bones/V	TW/V	Weak/V	LSR/V	M	M/R
2d-strip-packing	+++	+++		–		++	– – –	– –
argumentation	++	– – –		+++	–	–	+++	+++
bio			+++					
crypto-aes	++							
crypto-des				– –	+	++	+	++
crypto-gos		+					+	++
crypto-md5								
crypto-sha								
crypto-vmpc	++		–	++			– –	– –
diagnosis							–	
hardware-bmc						+++	– – –	– –
hardware-bmc-ibm				+			++	+++
hardware-cec								
hardware-manolios	+			– –			–	– –
hardware-velev		+		–		– –		–
planning					– –			
scheduling								
scheduling-pesp							–	–
software-bit-verif						+		
termination					–	+		

Table 4.5: Pearson (top) and Spearman (bottom) correlations between measures and MapleCOM-SPS solving time. Omits entries with less than 10 data points. Blue cells with a single '+' indicate moderate positive correlations ($0.4 \leq r < 0.6$); two '+' symbols indicates $0.6 \leq r < 0.8$; three '+' symbols indicates $r > 0.8$. Red '–' cells indicate negative correlations using the same system.

search for parameters that are more explanatory in this context.

We also remark that previous work showed notably higher R^2 values for community-based features [Newsham et al., 2014]. There are several significant differences between our experiments. First, our instances are pre-simplified using the MiniSat pre-processor before computing community structure. Their experiments grouped all application, crafted, and random instances into a single benchmark, whereas ours are split.

From the correlations results for each sub-category, it is clear that no measure correlates well across the board. Certain sub-categories, such as *argumentation*, appear to scale well with many parameters, whereas *e.g. hardware-**, *planning*, and *scheduling* instances do not correlate well with any of the considered measures. Further, certain measures may strongly positively correlate with solving time for some categories while strongly negatively correlating in others. For example, in Table 4.5, *M/R* has a 0.94 Spearman correlation with solving time over *argumentation* instances, whereas it has a -0.73 correlation for *hardware-manolios*. Thus it is not surprising that our regression R^2 results from above are not particularly strong for the full class of application instances. We further note that the mergeability-based features M and M/R tend to moderately or strongly correlate with many of the sub-categories, although the polarity of the correlation is dependent on the benchmark. This motivates our further empirical study of the measure in Chapter 5.

4.3 Classifying Benchmark Instances

A common use of structural measures of SAT formulas is to classify instances into sub-categories, such that portfolio-based solvers can choose the best algorithm to solve a given type of instance. More concretely, if we had a portfolio of SAT solvers such that one solver was tuned for each class of instances, then if we could correctly classify a given instance, we could choose the best algorithm to solve it fastest.

In [Ansótegui et al., 2017], it was shown that a small set of structural features based on community structure and fractal dimension were comparable to the set of 115 features used in SATzilla [Xu et al., 2008] when classifying sub-categories of the application track of the 2013 SAT competition. Many classification algorithms were considered, such as random forests or logistic regression, and the best algorithms achieved approximately 90% correct classification.

In this section, we repeat the experiments of [Ansótegui et al., 2017] with a significantly larger set of instances and more feature sets. We include all 795 instances from the application track of the SAT competitions from 2009-2014 that we could accurately place in the 19 sub-categories, as discussed above. Among the considered classification algorithms used in [Ansótegui et al., 2017],

	RF	LR	IBk	K*
$V \cdot C \cdot CVR$	64.512	33.5868	68.0608	45.2471
$V \cdot C \cdot Cmtys \cdot Q$	**74.1445**	49.3029	57.5412	45.6274
$V \cdot C \cdot \alpha_V \cdot dim_{VIG} \cdot dim_{CVIG}$	**88.0862**	61.4702	**84.9176**	46.2611
$V \cdot C \cdot Cmtys \cdot Q \cdot \alpha_V \cdot dim_{VIG} \cdot dim_{CVIG}$	**89.6071**	66.9202	67.3004	47.1483
$V \cdot C \cdot TW$	**76.9328**	53.6122	**70.9759**	45.6274
$V \cdot C \cdot LSR$	66.6667	44.4867	26.109	43.346
$V \cdot C \cdot Merges \cdot Res$	**74.5247**	50.3169	54.2459	29.4043
All Features	**90.8745**	**75.6654**	24.3346	28.7706

Table 4.6: Percentage of instances correctly classified by each algorithm and set of features.

we considered the top four performing algorithms: *random forests (RF), logistic regression (LR), k-nearest-neighbor (IBk),* and K^*. For each set of features, we constructed a classifier using 10-fold cross validation over the set of 795 instances. An important difference between our results and [Ansótegui et al., 2017] is that we pre-simplify all instances using MiniSat's pre-processing algorithm before collecting feature data. All classifiers were constructed with default parameters in Weka [Hall et al., 2009], as in [Ansótegui et al., 2017]. Table 4.6 displays the results. We highlight all classifiers that labelled at least 70% of instances correctly.

As in [Ansótegui et al., 2017], the random forest algorithm tends to produce the best classifiers. Interestingly, the self-similarity features (dim_{VIG} and dim_{CVIG}) on their own produced very strong classification rates (88.1%), which were only slightly improved by the community structure features. Other feature sets produced notably lower classification rates, although given that there were 19 sub-categories, classification rates were not very low, and most feature sets had over 70% correctness using random forests. Further, from a practical perspective, the random forest models, IBk, and K^* were built quite quickly (typically 10-fold cross validation took less than 10 seconds), however logistic regression required about 10 minutes for each feature set.

Structural Parameters for Industrial vs. Crafted Instances: The research question we posed here is the following: *Do real-world SAT instances have significantly more favorable parameter values (e.g. smaller backdoors or higher modularity), when compared to crafted or random instances?* A positive result, such that instances from application domains (including agile) have better structural values, would support the hypothesis that such structure may relate to the efficiency of CDCL solvers. Table 4.7 summarizes our results. We note that while application and agile instances indeed appear more structured with respect to these parameters, the application benchmark has high standard deviation values. This is likely due to the application instances coming from a wide variety of sources.

Benchmark	LSR/V	Weak/V	Q	Bones/V	TW/V
Agile	0.18 (0.13)	0.01 (0.01)	0.82 (0.07)	0.17 (0.11)	0.16 (0.08)
Application	0.35 (0.34)	0.03 (0.05)	0.75 (0.19)	0.64 (0.38)	0.32 (0.22)
Crafted	0.58 (0.35)	0.08 (0.11)	0.58 (0.24)	0.39 (0.41)	0.44 (0.29)
Random	0.64 (0.32)	0.11 (0.10)	0.14 (0.10)	0.47 (0.40)	0.82 (0.12)

Table 4.7: Mean (std. dev.) of several parameter values.

4.4 Evaluate Solving Heuristics via Structural Measures

When comparing different solvers or heuristics, the most common approach is to run them on a benchmark to compare solving times, or in some cases the number of conflicts during solving. However, such an approach does not lend much insight as to *why* one heuristic is better than another, nor does it suggest any ways in which a less performant heuristic may be improved. We discuss three sets of experiments that may help characterize the behavior of the SAT solver, and further allow us to compare different solving heuristics.

First, we consider locality-based experiments based on the underlying community structure of the instance (which partitions the variables into communities). We define two notions of locality: *spatial locality* describes the solver's behaviour of disproportionately branching upon variables from a small subset of communities, and *temporal locality* describes how the solver tends to branch upon variables in communities that have recently been branched upon. We show that modern branching heuristics are significantly more local than a baseline random branching heuristic.

Second, we discuss experiments that relate to the backbone of satisfiable instances, and consider how much work could be saved if the solver had *a priori* knowledge of the backbone. Finally, we consider the locality of the solver run as it relates to LSR backdoors, as discussed further in Section 6.5. Briefly, the computed LSR backdoor of an unsatisfiable instance is the set of all variables that occur in *useful* learned clauses during search (i.e. clauses that actually helped to derive unsat). The definition can be further adapted to satisfiable instances. A run of the solver that focuses on fewer variables can be seen as being more local, which may be favorable in terms of runtime on certain instances.

Starting from the baseline MapleSAT solver [Liang et al., 2016b], built on top of MiniSat [Eén and Sörensson, 2003], we consider three branching heuristics and three restart policies, totaling nine configurations. For the branching heuristics, we used LRB, a recent heuristic based on the notion of learning rate [Liang et al., 2016b], VSIDS [Moskewicz et al., 2001], and random

variable selection. We used phase-saving for polarity selection in all three cases. The three restart heuristics we considered were: 1) the default MapleSAT heuristic based on the Luby sequence [Luby et al., 1993]; 2) restarting after every conflict (always restart); and 3) never restarting. Due to the sheer number of SAT runs that must be evaluated, we only consider the full set of agile instances, and a subset of application instances. For the application instances, we first considered the subset of instances that MapleCOMSPS could solver in under 5000 seconds. From this set, we randomly selected 100 instances from the 2009 to 2014 SAT competitions, of which 50 were satisfiable and 50 unsatisfiable. The complete list of 100 instances is given in Appendix B. For runtime computation, we used the same computing clusters as described in Section 4.2. For every experiment we include an additional table over application instances, categorized by benchmark sub-category, however we only consider LRB branching and the 3 restart policies. We allotted 6 GB of memory for each run, and used a time cutoff of 5000 seconds for SAT competition instances, and 60 seconds for agile instances. All other data in the following subsections were computed using an additional run of the solver which performed necessary logging.

Figure 4.1 depicts the cactus plots for the nine solver configurations over the two benchmarks. For the application benchmark, LRB with Luby restarts is able to solve all 100 instances within the timeout, and VSIDS with Luby solved 88. LRB also outperforms VSIDS regardless of the restart heuristic configuration on this benchmark. Luby restarts outperformed the other two restart strategies, solving between 5 and 10 more instances when the branching heuristic is fixed. For the agile benchmark, VSIDS with Luby restarts solves approximately 150 more instances than LRB with Luby. The Luby restart policy outperforms the other policies when the branching heuristic remains constant. Random branching performs significantly worse than the other heuristics, as expected. In both benchmarks, the branching heuristic appears to be more important to performance than the restart policy. In the application benchmark, LRB outperformed VSIDS regardless of the restart policy, whereas VSIDS outperformed LRB on the agile benchmark.

Finally, we note a few interesting observations if we split the agile benchmark instances based on satisfiability, as depicted in Figure 4.2. If we only consider the unsatisfiable instances, the always-restart strategy performs much better; the LRB with always-restart strategy performs the best of all strategies, solving 4 more instances than VSIDS with Luby. However, the always-restart policy performs much worse than Luby on satisfiable instances, solving approximately 30% fewer instances. Conversely, never restarting performs remarkably well on satisfiable instances, solving approximately 25% more instances than Luby when fixing the branching heuristic. However, never restarting performs poorly on unsatisfiable instances, solving 10% fewer instances when fixing the branching heuristic to be VSIDS, and 25% fewer if we use LRB. This suggests that while Luby restarts do not perform the best when restricted to only satisfiable or only unsatisfiable instances, it does strike a good balance between the more extreme restart policies, and performs the best overall.

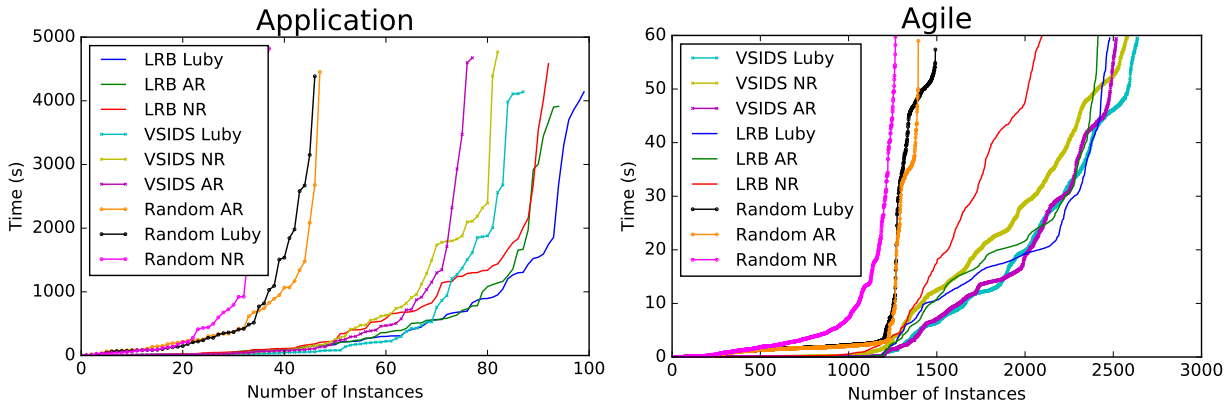


Figure 4.1: Cactus plots for the 9 solver configurations over the two benchmarks. The legends are ordered best to worst. Abbreviations: AR – always restart policy; NR – never restart.

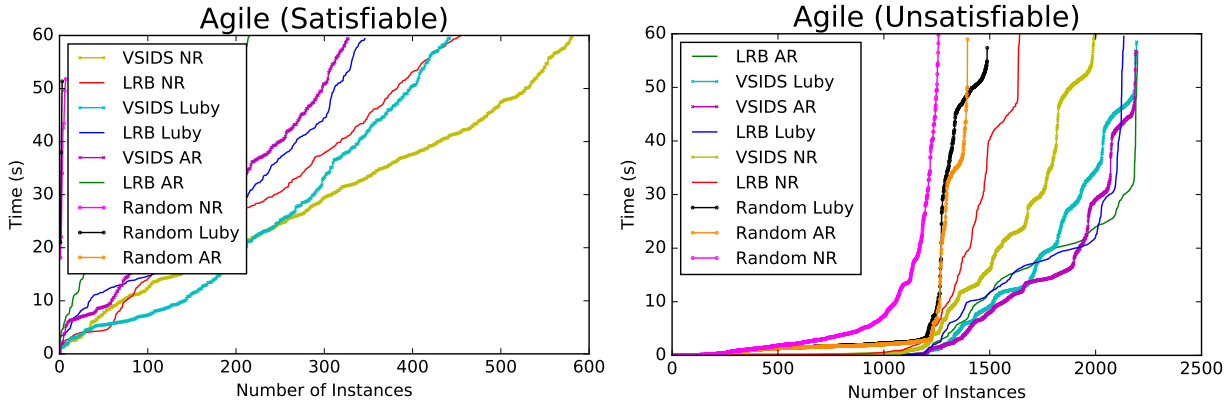


Figure 4.2: Cactus plots for the agile instances, split by satisfiability.

4.4.1 Community Structure Locality

Here, we use the community structure to assess the locality of various branching heuristics.

Spatial Locality. Our first hypothesis is that modern branching heuristics will exhibit *spatial locality* by disproportionately picking variables from a small subset of communities. For each instance, we first compute the community structure of the VIG. We then solve the instance using MapleSAT, however any time the solver makes a decision, the community of the chosen variable is recorded. For each community c, let p_c be the number of times the solver chose a variable from c, and let v_c be the number of variables in c. The normalized "score" associated with c is p_c/v_c.

Then, one can compute the Gini coefficient [Gini, 1921], a statistical measure of inequality,

Heuristic		Application		Agile	
Branching	**Restart**	**Gini Picks**	**Gini Clauses**	**Gini Picks**	**Gini Clauses**
LRB	Luby	0.50	0.54	0.66	0.71
	Always	0.50	0.54	0.67	0.70
	Never	0.54	0.55	0.72	0.72
VSIDS	Luby	0.52	0.56	0.64	0.70
	Always	0.52	0.56	0.66	0.71
	Never	0.56	0.56	0.74	0.73
Random	Luby	0.16	0.48	0.18	0.60
	Always	0.15	0.50	0.16	0.64
	Never	0.18	0.47	0.21	0.57

Table 4.8: Measures the spatial locality of the branching heuristics' decisions ("Gini Picks" in Table 4.8), with respect to the underlying community structure. Further measures a similar locality notion of the learned clauses.

over these scores to measure how disproportionately the solver picks from certain communities. The value ranges from 0 to 1, where 0 indicates total equality, and 1 indicate total disparity (i.e. all picks from a single community). The intuition behind this experiment and the use of the Gini coefficient here (used in measuring the inequality of wealth distribution in countries) is that it is an effective method for computing how unequally a branching heuristic favors some communities over others. Using this metric we show for example that VSIDS disproportionately favors a small set of communities (highly unequal distribution of picks) compared to the random branching heuristic (largely equal distribution of picks). The results are associated with the "Gini Picks" columns in Table 4.8 (averaged over all instances in the benchmark), and data aggregated by sub-category is in Table 4.9.

As a followup experiment, we also compute a measure of the locality of learned clauses produced by the solver. For any learned clause of length n, for each literal in the clause, we increment the score of its associated community by $1/n$. After the solver finishes, we normalize the community scores by their sizes, and compute the Gini coefficient again. This is listed under the "Gini Clauses" columns of Table 4.8.

Results suggest that VSIDS appears to choose variables much more locally than random, and LRB performs similarly. The restart policy appears to have little affect on locality here, although never restarting seems to result in slightly more locality. All heuristics tend to be much more local on the agile instances. In regard to the locality of the clauses that the solver learns during search, although LRB and VSIDS appear more local, the difference compared to random branching is

Category	Q	Cmtys	Luby	Always	Never
2d-strip-packing (14)	0.89 (0.05)	30.14 (8.55)	0.45 (0.17)	0.47 (0.18)	**0.60 (0.21)**
argumentation (16)	0.05 (0.02)	3.94 (1.12)	0.48 (0.22)	0.48 (0.21)	**0.49 (0.20)**
bio (40)	0.56 (0.11)	29.60 (14.35)	0.46 (0.18)	0.48 (0.20)	**0.60 (0.19)**
crypto-aes (23)	0.73 (0.09)	22.96 (11.85)	0.52 (0.16)	0.53 (0.15)	**0.56 (0.16)**
crypto-des (12)	0.92 (0.00)	80.50 (2.91)	0.46 (0.04)	0.46 (0.04)	**0.47 (0.04)**
crypto-gos (31)	0.68 (0.07)	35.06 (18.81)	**0.80 (0.26)**	0.79 (0.27)	0.79 (0.27)
crypto-md5 (15)	0.81 (0.03)	35.13 (11.39)	0.28 (0.06)	0.29 (0.08)	**0.32 (0.09)**
crypto-sha (45)	0.68 (0.01)	14.42 (0.97)	0.08 (0.05)	0.10 (0.05)	**0.33 (0.12)**
crypto-vmpc (21)	0.21 (0.01)	6.86 (0.91)	**0.12 (0.05)**	0.11 (0.04)	0.10 (0.05)
diagnosis (80)	0.92 (0.02)	56.66 (20.32)	0.51 (0.13)	**0.52 (0.13)**	**0.52 (0.13)**
fpga-routing (2)	0.59 (0.00)	12.50 (0.71)	**0.92 (0.01)**	**0.92 (0.01)**	**0.92 (0.00)**
hardware-bmc (18)	0.88 (0.06)	22.83 (10.33)	0.29 (0.12)	0.28 (0.13)	**0.34 (0.13)**
hardware-bmc-ibm (13)	0.91 (0.03)	28.46 (7.37)	0.31 (0.18)	0.33 (0.20)	**0.63 (0.22)**
hardware-cec (26)	0.77 (0.09)	28.31 (18.85)	0.42 (0.18)	0.42 (0.18)	**0.50 (0.19)**
hardware-manolios (22)	0.77 (0.04)	19.05 (4.96)	0.52 (0.13)	0.50 (0.12)	**0.59 (0.16)**
hardware-velev (18)	0.64 (0.08)	10.61 (7.88)	0.26 (0.07)	0.22 (0.07)	**0.37 (0.11)**
planning (13)	0.85 (0.03)	29.23 (15.78)	0.59 (0.23)	**0.60 (0.26)**	0.58 (0.21)
scheduling (50)	0.88 (0.04)	38.50 (8.12)	0.69 (0.16)	0.69 (0.15)	**0.80 (0.12)**
scheduling-pesp (26)	0.79 (0.04)	14.27 (11.72)	0.86 (0.13)	**0.87 (0.12)**	0.85 (0.15)
software-bit-verif (40)	0.75 (0.10)	15.72 (10.87)	0.43 (0.19)	0.41 (0.20)	**0.45 (0.19)**
software-bmc (4)	0.97 (0.01)	87.00 (21.34)	0.63 (0.15)	0.63 (0.15)	**0.73 (0.11)**
termination (38)	0.79 (0.08)	48.74 (39.51)	0.47 (0.20)	**0.48 (0.21)**	**0.48 (0.21)**

Table 4.9: Mean (std. dev.) of the Gini coefficient of variable picks, grouped by formula category. The number after the category name is the number of instances considered.

much less significant than in the "picks" case. This suggests that even though random branching is less focused on certain communities during branching, the learned clauses derived still tend to be focused on a similar number of communities to the other branching heuristics.

Temporal Locality. Our next hypothesis is that modern branching heuristics exhibit *temporal locality* by picking variables from communities that were recently chosen from. Define our window size *ws* to be *n*% of the total number of communities ($n \in \{1, 10, 25\}$ for our experiments), rounded up to the nearest integer. For all instances, our window contains the set of communities from the *ws* most recent decisions (note that the set may have less than *ws* elements). At every decision, we increment a counter *window_hits* if the current variable is from a community in the window. We assign a temporal score $ts = window\_hits/decisions$ for each run of the solver. We report the

Heuristic		Application		
Branching	**Restart**	**Window 1**	**Window 10**	**Window 25**
LRB	Luby	0.44 (0.21)	0.53 (0.21)	0.66 (0.20)
	Always	0.43 (0.22)	0.51 (0.22)	0.64 (0.21)
	Never	0.45 (0.21)	0.54 (0.21)	0.67 (0.20)
VSIDS	Luby	0.48 (0.21)	0.59 (0.22)	0.71 (0.22)
	Always	0.48 (0.21)	0.58 (0.21)	0.70 (0.21)
	Never	0.47 (0.21)	0.59 (0.22)	0.71 (0.21)
Random	Luby	0.13 (0.08)	0.19 (0.09)	0.33 (0.10)
	Always	0.12 (0.08)	0.19 (0.09)	0.32 (0.10)
	Never	0.13 (0.08)	0.20 (0.10)	0.33 (0.11)

Table 4.10: Temporal locality for the application benchmark with varying window sizes.

Heuristic		Agile		
Branching	**Restart**	**Window 1**	**Window 10**	**Window 25**
LRB	Luby	0.50 (0.25)	0.61 (0.20)	0.75 (0.14)
	Always	0.47 (0.27)	0.57 (0.22)	0.71 (0.16)
	Never	0.53 (0.22)	0.65 (0.18)	0.80 (0.13)
VSIDS	Luby	0.55 (0.22)	0.68 (0.18)	0.81 (0.13)
	Always	0.53 (0.23)	0.64 (0.19)	0.78 (0.14)
	Never	0.52 (0.23)	0.66 (0.19)	0.82 (0.14)
Random	Luby	0.11 (0.07)	0.19 (0.07)	0.37 (0.08)
	Always	0.10 (0.06)	0.17 (0.07)	0.35 (0.08)
	Never	0.13 (0.08)	0.21 (0.09)	0.39 (0.09)

Table 4.11: Temporal locality for the agile benchmark with varying window sizes.

average ts value for each benchmark category in Tables 4.10 and 4.11.

The key idea behind this experiment is to test the hypothesis that modern branching heuristics favor picking from recently picked-from communities, versus random which does not display such temporal locality. This is clearly the case from our experiments: even with a small window of 1% of the communities, both LRB and VSIDS pick from these recent communties approximately 45-55% of the time, whereas the random branching heuristic has around 12% hits. The restart policy does not appear to have as much of an effect, however the always-restart policy tends to be

less local than others.

4.4.2 Backbone-based Locality

Several previous works have considered practical applications of computing backbones. In the interactive product configuration domain, the backbone is used to restricts the user from choosing invalid configurations [Batory, 2005]. Backbone-based heuristics have also been used in pseudo-Boolean solving [Manolios and Papavasileiou, 2011] and quantified Boolean formula (QBF) solving [Lonsing and Biere, 2011]. In general, although computing backbones can be quite expensive, it may be useful for problem domains which require many SAT solver calls on the same instance.

We test two ways in which a default solver (i.e. with no a priori information regarding the backbone), is performing work that a solver with knowledge of the backbone would not have to perform. For each default solver (with varying branching heuristics and restart policies), we record the number of clauses that are subsumed by the backbone, i.e., the set of clauses that have any literal with the same polarity as in the backbone. These clauses would be redundant if the solver had *a priori* knowledge of the backbone, since they would be immediately satisfied by the backbone. Put differently, if the solver were additionally given the backbone as a set of unit clauses, these learned clauses would be trivially subsumed by the unit clauses, and could never be learned by the solver.[2] We report our results in two ways in Table 4.12. First, note that the average backbone size in our application benchmark is 63% of the total variables, and 18% for the agile instances. The "Subs/L" columns indicate the number of learned clauses subsumed by the backbone, divided by the total learned clauses. The "Subs/B" columns normalizes the number of subsumed clauses by the size of the backbone. In Table 4.13, we report the "Subs/L" values aggregated by sub-category.

In our second experiment, we test how often the polarity of backbone literals are "flipped" during search. This may, in some sense, characterize that the solver is either focused on these literals, or is unsure of their correct polarity. Any time a backbone literal is placed on the trail, we record its polarity. If the opposite polarity literal is placed the trail at a later time, we increment a counter. For a given solver run, we normalize this counter by the number of backbone literals in the instance (denoted by "Flips/B" in Table 4.12). For example, in the case of LRB branching with Luby restarts, for an average application instance, 68% of the learned clauses are subsumed by

[2]This relates to the notion of clause absorption [Atserias et al., 2011]. Informally, a clause C is absorbed by a set of clauses Δ if the C does not add any "propagation power" to Δ, i.e., adding C to Δ does not allow the unit propagator to perform any additional propagations.

Heuristic		Application (63% backbone)			Agile (17% backbone)		
Branching	**Restart**	**Subs/L**	**Subs/B**	**Flips/B**	**Subs/L**	**Subs/B**	**Flips/B**
LRB	Luby	0.68	5700	165933	0.03	1	25
	Always	0.68	3673	94888	0.03	2	34
	Never	0.68	5492	163923	0.05	4	99
VSIDS	Luby	0.68	13530	132655	0.03	2	59
	Always	0.68	9002	47373	0.03	2	38
	Never	0.67	12997	177712	0.06	11	226
Random	Luby	0.69	16396	362989	0.05	10	209
	Always	0.68	11611	284923	0.03	7	183
	Never	0.69	17695	380419	0.11	22	379

Table 4.12: Measures how often the solver learns clauses that would be subsumed by the backbone. Further measures how many times the polarity of backbone literals are flipped during solving. Abbreviations: Subs/L: the number of learned clauses subsumed by the backbone, normalized by dividing by the number of learned clauses; Subs/B: the number of learned clauses subsumed by the backbone, normalized by the size of the backbone; Flips/B: the number of times the solver changes the polarity of a backbone literal, normalized by the size of the backbone.

the backbone, the solver learns 5,700 clauses subsumed by the backbone per backbone variable, and on average flips a backbone literal's polarity 165,933 times.

When considering the *subsumed/learned* ratio, the differing heuristics appear to have little effect. The always-restart heuristic avoids more backbone-subsumed clauses (as indicated by the *subsumed/backbones* ratios), likely since it requires less conflicts to solve the instance. The number of times backbone variables get flipped also seem to occur proportionally to this value. Results aggregated by sub-category in Table 4.13 appear consistent with this observation.

It may be worth comparing how this affects performance on each of the heuristics, if the solver was given the backbone in advance. Nonetheless, even if the solver is given the backbone in advance (and would thus avoid the work described in Table 4.12), there is no guarantee that the solver would actually run faster, due to its heuristic nature.

4.4.3 LSR Backdoor Locality

Here, we provide a measure of the locality of solver runs, based on the notion of LSR backdoors, using the approach to compute them in Section 6.5. We emphasize that although backdoors are

Category	Backbones	Luby	Always	Never
2d-strip-packing (1)	0.43 (–)	0.42 (–)	0.43 (nan)	0.39 (–)
argumentation (1)	0.84 (–)	0.92 (–)	0.94 (nan)	0.90 (–)
bio (14)	0.20 (0.30)	0.29 (0.35)	0.28 (0.34)	0.27 (0.37)
crypto-aes (10)	0.76 (0.33)	0.97 (0.05)	0.97 (0.06)	0.95 (0.12)
crypto-des (12)	1.00 (0.00)	1.00 (0.00)	1.00 (0.00)	1.00 (0.00)
crypto-md5 (1)	0.00 (–)	0.00 (–)	0.00 (–)	0.00 (–)
crypto-sha (9)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
crypto-vmpc (18)	0.96 (0.04)	1.00 (0.00)	1.00 (0.00)	1.00 (0.01)
diagnosis (27)	0.89 (0.03)	0.87 (0.09)	0.87 (0.09)	0.88 (0.09)
hardware-bmc-ibm (3)	0.23 (0.22)	0.21 (0.34)	0.20 (0.33)	0.25 (0.41)
planning (2)	0.39 (0.44)	0.50 (0.59)	0.55 (0.54)	0.54 (0.51)
scheduling (3)	0.13 (0.22)	0.28 (0.49)	0.28 (0.48)	0.26 (0.45)
scheduling-pesp (1)	0.00 (–)	0.00 (–)	0.00 (–)	0.00 (–)
software-bit-verif (6)	0.20 (0.39)	0.22 (0.39)	0.23 (0.39)	0.23 (0.39)
termination (26)	0.48 (0.28)	0.55 (0.31)	0.55 (0.32)	0.55 (0.31)

Table 4.13: Mean (std. dev.) of the fraction of variables in the backbone, and fraction of learned clauses subsumed by the backbone ("Subs / L" in Table 4.12), grouped by formula category. The number after the category name is the number of instances considered.

typically defined for a SAT formula (rather than a run of the solver), we can use the concept to draw comparisons between runs. For example, in the case of an unsatisfiable formula, we are measuring the size of the set of variables found in the learned clauses of the proof (clauses that are not used in the proof are ignored). As discussed further in Section 6.5, a fresh solver would be able to derive UNSAT by only branching on the variables in this set. For each solver run, we compute the LSR backdoor size, and then divide by the total number of variables. Aggregated results over each benchmark are reported in Table 4.14. We only report data on instances for which we could compute LSR backdoors for all nine heuristics. Results for each benchmark sub-category are listed in Table 4.15.

We find that restarting after every conflict tends to produce the smallest LSR backdoors (when fixing the branching heuristic), indicating that in some sense the solver is finding a more local proof. This is consistent across most sub-categories. We conjecture that this behaviour relates to previous work that showed rapid restarting solvers required fewer conflicts during solving [Haim and Heule, 2014].

Branching	Restart	Application LSR	Agile LSR
LRB	Luby	0.39	0.19
	Always	**0.36**	**0.14**
	Never	0.39	0.31
VSIDS	Luby	0.40	0.21
	Always	**0.34**	**0.14**
	Never	0.36	0.30
Random	Luby	0.50	0.70
	Always	**0.34**	**0.27**
	Never	0.51	0.70

Table 4.14: LSR backdoor comparison of the proofs/models found by each heuristic. Each cell indicates the average ratio of LSR backdoor variables over total variables for instances in the benchmark. We highlight the ratios in the always-restart rows, as this heuristic consistently produces smaller ratios.

4.5 Threats to Validity

Due to the difficulty of computing the exact/best values for the parameters considered, we must rely upon heuristic algorithms. We attempt to mitigate this threat by using the best off-the-shelf algorithms for each parameters and by allowing long runtimes when necessary. The values associated with weak backdoor and LSR backdoor sizes were also generated with a run of a solver. Since we are correlating versus solver runtime, this may add some "additional correlation" to the regression results, however we note that different solvers were used for time and backdoor computation. Our results are also dependent upon the instances that we studied. We attempt to make our results more generally applicable, by studying a large set of instances from multiple categories of the SAT competitions. We mitigate this by separating the results with respect to the SAT competition categories, and further by considering each sub-category. We also only used a small set of random instances which may affect results, however given the intrinsically unstructured nature of random instances, we expect that adding more instances would only make correlations worse. We only used a single run of the random branching heuristic for each instance in Section 4.4 due to resource constraints. Our results are dependent on the solvers MapleSAT, MapleCOMSPS, and Lingeling. We expect similar results for other MiniSAT-based solvers, but more solvers could be considered.

Category	Luby	Always	Never
2d-strip-packing (7)	0.49 (0.26)	**0.46 (0.27)**	0.51 (0.24)
bio (7)	**0.77 (0.34)**	0.80 (0.35)	**0.77 (0.34)**
crypto-aes (2)	0.50 (0.03)	**0.48 (0.09)**	0.49 (0.08)
fpga-routing (3)	**0.02 (0.00)**	**0.02 (0.00)**	0.07 (0.05)
hardware-manolios (2)	0.49 (0.25)	**0.44 (0.27)**	0.46 (0.24)
hardware-velev (2)	0.70 (0.04)	**0.69 (0.07)**	0.70 (0.02)
planning (2)	0.41 (0.10)	**0.29 (0.08)**	0.40 (0.09)
scheduling-pesp (2)	**0.30 (0.06)**	0.32 (0.08)	0.32 (0.02)
software-bit-verif (12)	0.93 (0.09)	**0.85 (0.13)**	0.92 (0.12)
software-bmc (1)	0.10 (–)	**0.09 (–)**	0.10 (–)
termination (6)	0.95 (0.06)	**0.93 (0.08)**	0.95 (0.06)

Table 4.15: Mean (std. dev.) of LSR backdoor size, grouped by formula category. The number after the category name is the number of instances considered.

4.6 Related Work

Backdoor-related Parameters

Traditional weak and strong backdoors for both SAT and CSP were introduced by Williams et al. [Williams et al., 2003a]. Kilby et al. introduced a local search algorithm for computing weak backdoors [Kilby et al., 2005], and showed that random SAT instances have medium sized backdoors (of roughly 50% of the variables), and that the size of weak backdoors did not correlate strongly with solving time. Li et al. introduced an improved Tabu local search heuristic for weak backdoors [Li and Van Beek, 2011]. They demonstrated that many industrial satisfiable instances from SAT competitions have very small weak backdoors, often around 1% of the variables. The size of backdoors with respect to subsolvers different from UP was considered in [Dilkina et al., 2014, Li and Van Beek, 2011]. Monasson et al. introduced backbones to study random 3-SAT instances [Monasson et al., 1999]. Janota et al. [Janota et al., 2015] introduced and empirically evaluated several algorithms for computing backbones. Several extensions of traditional strong and weak backdoors have been proposed. LS backdoors also consider the assignment tree of backdoor variables, but additionally allow clause learning to occur while traversing the tree, which may yield exponentially smaller backdoors than strong backdoors [Dilkina et al., 2009a, Dilkina et al., 2009b].

Graph Abstraction Parameters. Mateescu computed lower and upper bounds on the treewidth of large application formulas [Mateescu, 2011]. Ansótegui et al. introduced com-

munity structure abstractions of SAT formulas, and demonstrated that industrial instances tend to have much better structure than other classes, such as random [Ansótegui et al., 2012]. It has also been shown that community-based features are useful for classifying industrial instances into subclasses (which distinguish types of industrial instances in the SAT competition) [Jordi, 2015]. Community-based parameters have also recently been shown to be one of the best predictors for SAT solving performance [Newsham et al., 2014].

Other work such as SatZilla [Xu et al., 2008] focus on large sets of easy-to-compute parameters that can be used to quickly predict the runtime of SAT solvers. In this paper, our focus is on parameters that, if sufficiently favorable, offer provable parameterized complexity-theoretic guarantees of worst-case runtime [Downey and Fellows, 2013]. The study of structural parameters of SAT instances was inspired by the work on clause-variable ratio and the phase transition phenomenon observed for randomly-generated SAT instances in the late 1990's [Coarfa et al., 2000, Monasson et al., 1999, Selman et al., 1996].

Table 4.1 lists previous results on empirically computing several parameters and correlating them with SAT solving time. While weak backdoors, backbones, and treewidth have been evaluated on some industrial instances from the SAT competitions, only modularity has been evaluated across a wide range of instances. Some benchmarks, such as the random 3-SAT and graph coloring instances considered in [Kilby et al., 2005], are too small/easy for modern day CDCL solvers to perform meaningful analysis. Additionally, the benchmarks used in previous works to evaluate each parameter are mostly disjoint, making comparisons across the data difficult.

4.7 Conclusions

Motivated by the remarkable performance of CDCL SAT solvers on industrial instances, we performed a large-scale comprehensive study of several well-known structural parameters of SAT instances and their correlations with solver runtime over a diverse and representative set of 7000+ SAT competition instances. We found that while most of these features correlate with solving time for certain classes of formulas, these correlations were not strong for the entire benchmark suite we studied.

Further, these measures can be used as a lens to analyze the behavior of SAT solvers and how they explore the search space. We showed that the locality of the solver, with respect to these measures, is often highly dependent upon the underlying branching heuristic and restart policy. While it is clear from our results that modern branching heuristics exhibit much more locality than a baseline randomized solver, it remains unclear just "how much" a solver should remain local during search. For example, when considering community structure-based locality, the

never-restart policy is significantly more local than the other heuristics, however this is generally not a good restart policy in practice. Given that the Luby restart policy generally performs the best on average, our results suggest that heuristics that promote some locality, while also allowing the solver to occasionally explore new regions of the search space, perform the best in general.

Chapter 5

Merge Resolutions and CDCL Solving

From our results in Chapter 4, mergeability tended to correlate with solving time for many sub-categories of application instances (cf. Table 4.5). This chapter discusses an empirical study of mergeability, which quantifies how many pairs of input clauses are mergeable. Two clauses are mergeable if they resolve and share a common literal. Merge resolutions are particularly important, as they allow the resolvent clause to be smaller than the two clauses being resolved. Thus, the theme of this chapter revolves around the following question: *If one scales Boolean formulas to have more mergeable clause pairs (while keeping other formula characteristics invariant), how does this affect the performance of CDCL SAT solvers?*

Main Contributions:

- We describe an algorithm for generating increasingly mergeable formulas, given a *seed* formula. We prove that our approach maintains several key properties of the original instance, such as the distribution of variable occurrences, and properties of the underlying community structure.
- We derive the expected number of merges a random-kSAT instance is expected to have for varying k, number of variables (n), and number of clauses (m).
- We empirically evaluate the importance of mergeability by considering a set of industrial-like randomly-generated instances, and creating series of increasingly more mergeable instances. We show that mergeability strongly negatively correlates with CDCL solving time over sets of unsatisfiable instances. We further show that the solver tends to on average produce shorter learned clauses as we scale up the number of mergeable clauses in randomly-generated instances.

Algorithm 2 Greedy Approach to Increasing Mergeable Input Clauses

input: CNF Formula F
output: Modified CNF Formula F'
set *lockedLits, lockedClauses, flipPairs*
bool *formulaHasChanged* $\leftarrow$ `true`
while *formulaHasChanged* **do**
    lockedLits, lockedClauses, flipPairs $\leftarrow \{\}, \{\}, \{\}$
    formulaHasChanged $\leftarrow$ *false*
    for every pair of clauses c_i, c_j **do**
        if $(|c_i \cap \{\neg l | l \in c_j\}| \geq 1)$ **then**
            for every merge literal $l \in c_i \cap c_j$ **do** ▷ lock literals that merge
                $lockedLits \leftarrow lockedLits \cup \{(c_i, l), (c_j, l)\}$
            if $(|c_i \cap \{\neg l_1 | l_1 \in c_j\}| == 1) \wedge (\exists l_2 : l_2 \in c_i \cap c_j)$ **then**
                $lockedLits \leftarrow lockedLits \cup \{(c_i, l_1), (c_j, \neg l_1)\}$ ▷ lock literals that resolve mergeable clauses
            else if $(|c_i \cap \{\neg l | l \in c_j\}| > 1)$ **then**
                for every literal pair $l \in c_i$ and $\neg l \in c_j$ **do**
                    $flipPairs \leftarrow flipPairs \cup \{((c_i, l), (c_j, \neg l))\}$
    for $((c_i, l_m), (c_j, l_{\neg m})) \in flipPairs$ **do**
        if $c_i, c_j \notin lockedClauses \wedge (c_i, l_m), (c_j, l_{\neg m}) \notin lockedLits$ **then**
            if $\exists c_k \notin lockedClauses \wedge \neg l_m \in c_k \wedge (c_k, \neg l_m) \notin lockedLits$ **then**
                $flip(c_i, l_m)$
                $flip(c_k, \neg l_m)$
                $lockedClauses \leftarrow lockedClauses \cup \{c_i, c_j, c_k\}$
                $formulaHasChanged \leftarrow true$
            else if $\exists c_k \notin lockedClauses \wedge l_m \in c_k \wedge (c_k, l_m) \notin lockedLits$ **then**
                $flip(c_j, \neg l_m)$
                $flip(c_k, l_m)$
                $lockedClauses \leftarrow lockedClauses \cup \{c_i, c_j, c_k\}$
                $formulaHasChanged \leftarrow true$

5.1 Generating Mergeable Formulas

We propose a greedy approach to increase the number of merges, as described in Algorithm 2. Our approach works in the following main steps. We take as input an arbitrary formula in CNF. Then, up until fixed-point, we seek pairs of "tautologically resolvent" clauses, i.e., clauses that

resolve on two or more variables. Consider the following example clauses:

$$(x \vee y) \wedge (\neg x \vee \neg y). \tag{5.1}$$

This pair of clauses resolve on both x and y, and is therefore not mergeable nor resolvable.[1] However, if we flip the polarity of any of the four literals:

$$(x \vee y) \wedge (\neg x \vee y), \tag{5.2}$$

then the clauses both resolve (on x) and merge (on y). This process repeats until no more changes to the formula increase the number of overall merges. In order to ensure progress, an additional invariant ensures that if two clauses merge on the original instance, then they will also merge in the generated instance. While either of the clauses may be modified (by flipping other literals in the clauses to allow additional merges), a merge will still exist between them.

We represent literals as (clause_id, literal) pairs in order to distinguish different instances of the same literal. There are three main sets that ensure our invariants. The set *lockedLits* maintains all literals that are either merged by two resolvable clauses, or are the literals that are actually resolved in a merge. Suppose we have clauses $c_i = (a \vee b \vee c), c_j = (\neg a \vee b \vee d)$. Since the clauses resolve on a single variable a and merge on b, Line 11 will add (c_i, b) and (c_j, b) to *lockedLits*, and Line 13 will add (c_i, a) and $(c_j, \neg a)$. This ensures that the algorithm will never flip the polarity of any of these literals, and that the output formula contains these same merges. Suppose that two clauses, e.g., $c_i = (a \vee b \vee c), c_j = (\neg a \vee b \vee \neg c)$ contain multiple literals upon which to resolve (a and c). Then on Lines 15-16, we add all resolving pairs of literals to *flipPairs* (e.g. $((c_i, a), (c_j, \neg a))$ and $((c_i, c), (c_j, \neg c))$). The intuition is that if any one of these literal's polarities were flipped, then the two clauses would merge on two literals instead of one, while still being resolvable (e.g. $c_i = (a \vee b \vee \neg c), c_j = (\neg a \vee b \vee \neg c)$).

In Lines 17-27, we iterate through all pairs of literals in *flipPairs*. If the associated clauses and literals are not locked (Line 18), then we may try to flip one of the literals to increase the overall number of merges. In order to flip some literal, e.g., (c_i, l_m) to $(c_i, \neg l_m)$, there must exist some other unlocked literal $(c_k, \neg l_m)$ that we can flip, in order to ensure that the overall literal counts remain constant. If so, we flip the corresponding literals and lock the three involved clauses (Lines 20-23, and similarly on Lines 25-28). This ensures that we do not end up flipping too many literals in the same clause, to the point where the clauses no longer resolve. This process repeats until a fixpoint, clearing the three involved sets at the start of each iteration (Line 6). Note that in practice, we randomize the ordering in this data structure so that we do not prefer flipping literals in earlier clauses.

[1] Note that in the resolution proof system, any resolvent of these clauses (e.g. $(x \vee y \vee \neg y)$) is effectively useless since it is tautologically true.

5.1.1 Properties of the Generator

We designed our algorithm to retain as many properties of the original instance as possible, while increasing mergeability. The following are known properties retained by our approach: the number of variables and clauses, all community structure properties based on the variable incidence graph (i.e. modularity), popularity of variables and literals, and resolvability. The following are known properties that are *not* retained: satisfiability, and community structure properties if computed over the literal incidence graph. We formally discuss several of these properties in the following results.

Observation 2. Algorithm 2 terminates.

Proof. First, note that the only time that the formula is changed is in calls to *flip*(...) on Lines 20-21, 25-26. We show that any time these sets of Lines are invoked, the size of *lockedLits* must increase in the next iteration of the while-loop, and since the formula is finite, the algorithm will eventually terminate. Note that if *flip* is never invoked, then the while-loop must terminate as *formulaHasChanged* must be *false*.

We first show every element of *lockedLits* from the previous while-loop iteration will still be in *lockedLits* in the next iteration. There are two cases to consider. First, if two clauses resolve on a single literal and merge, then the resolving literal is locked (Line 13) and all merging literals are locked (Lines 10-11). Since these literals can then never be flipped, the same literals will be locked due to this clause pair in the next while-loop iteration. Further, flipping any of the unlocked literals in the clause pair will not reduce the number of merges (e.g. by creating a second pair of conflicting literals between the clauses, thus making the clause pair no longer resolvable). Second, if the clause pair has multiple conflicting literals (thus adding pairs to *flipPairs*), then only the literals that are merges are added to *lockedLits* (unless added by a different pair of clauses). Since a clause can only be changed on one literal at each iteration of the while-loop (due to *lockedClauses* being updated on Lines 22, 27), there will still exist at least one pair of conflicting literals to satisfy the if-statement on Line 9 in the next iteration.

Assume *w.l.o.g.* that we flip (c_i, l_m) and $(c_k, \neg l_m)$ on Lines 20-21. Then in the next iteration both c_i and c_j will contain the literal $\neg l_m$, and since they must still have a conflicting literal, $(c_i, \neg l_m)$ and $(c_j, \neg l_m)$ will be added to *lockedLits* on Line 11. □

Observation 3. Each iteration of the while-loop in Algorithm 2 (Lines 6-27) cannot decrease the mergeability of the formula.

Proof. If a clause pair already merges on (potentially several) literals and resolves, then all relevant literals will be locked and cannot be changed (through similar arguments as in Observation 2).

Thus, the mergeability will not decrease. As literals get flipped, it will decrease the number of conflicting literals between the clause pair, and increase the number of overlapping literals. If the number of conflicting literals is ever reduced to one, the number of overall merges will increase by the number of merge literals. □

Observation 4. Algorithm 2 does not preserve satisfiability.

Example 2. Consider the following formula over 7 variables and 9 clauses:

$$\begin{aligned}\mathscr{F}_{unsat} =&(\neg a \vee \neg b) \wedge (a \vee b) \wedge (b \vee c) \wedge \\ &(a \vee a_{h_1}) \wedge (a \vee a_{h_2}) \wedge (a \vee \neg a_{h_1} \vee \neg a_{h_2}) \wedge \\ &(b \vee b_{h_1}) \wedge (b \vee b_{h_2}) \wedge (b \vee \neg b_{h_1} \vee \neg b_{h_2})\end{aligned} \tag{5.3}$$

This formula is unsatisfiable. To see this, note that Line 2 alone of $\mathscr{F}_{unsat}$ is enough to ensure that a must be set to `true`, and similarly b must be set to `true` due ot Line 3. This in conjunction with $(\neg a \vee \neg b)$ is enough to make the formula unsatisfiable. When applying Algorithm 2, the first two clauses conflict on multiple literals, and none of the literals will be locked, as neither of the clauses can pair with any other clause to both resolve and merge. Further the literal b in $(b \vee c)$ is also unlocked. We can then flip either of the b literals in the first two clauses with the third clause:

$$\begin{aligned}\mathscr{F}_{sat} =&(\neg a \vee \mathbf{b}) \wedge (a \vee b) \wedge (\neg \mathbf{b} \vee c) \wedge \\ &(a \vee a_{h_1}) \wedge (a \vee a_{h_2}) \wedge (a \vee \neg a_{h_1} \vee \neg a_{h_2}) \wedge \\ &(b \vee b_{h_1}) \wedge (b \vee b_{h_2}) \wedge (b \vee \neg b_{h_1} \vee \neg b_{h_2})\end{aligned} \tag{5.4}$$

This formula is satisfiable under the assignment $\langle a \leftarrow \texttt{true}, b \leftarrow \texttt{true}, c \leftarrow \texttt{true} \rangle$, along with any assignment to the remaining variables.

Through similar reasoning, we can begin with the following satisfiable formula and transform it to be unsatisfiable:

$$\begin{aligned}\mathscr{G}_{sat} =&(\neg a \vee b) \wedge (a \vee \neg b) \wedge (b \vee c) \wedge \\ &(a \vee a_{h_1}) \wedge (a \vee a_{h_2}) \wedge (a \vee \neg a_{h_1} \vee \neg a_{h_2}) \wedge \\ &(b \vee b_{h_1}) \wedge (b \vee b_{h_2}) \wedge (b \vee \neg b_{h_1} \vee \neg b_{h_2})\end{aligned} \tag{5.5}$$

This formula is satisfiable under the same assignment as above. By flipping the b literals in $(a \vee \neg b)$ and $(b \vee c)$ we obtain an unsatisfiable formula.

Observation 5. Algorithm 2 can be modified to decrease mergeability by changing how lockedLits and flipPairs are computed on Lines 9-16.

If a pair of clauses has exactly 2 pairs of opposing literals (as in Equation 5.1), then we lock all 4 literals. Essentially, if one of these literals were to be flipped, the clause would resolve and merge. We do not lock any other literals. As for *flipPairs* (which now constitute pairs of literals which we hope to flip to *reduce* mergeability), if a clause pair merges, we add all pairs of merging literals, as well as the pair of resolving literals to the data structure. As we will show in the following subsection, for many uniform 3SAT formulas, we can reduce the number of mergeable pairs to zero.

5.2 Mergeability of Random-kSat Instances

Before discussing our empirical results, we demonstrate properties of randomly generated kSAT instances. Specifically, we find the expected number of merges a random-kSAT instance will have, which depends on the clause size k, the number of variables n, and the number of clauses m. This allows us to make useful comparisons among random-kSAT instances of varying size.

We assume that all clauses have exactly k variables, and each possible clause occurs with uniform probability.

Theorem 2. Let F be a random-kSAT formula with n variables and m clauses. Then the expected number of merges over input clause pairs is:

$$E(merges(F)) = \binom{m}{2} \sum_{i=1}^{k} \frac{i(i-1)}{2^i} \cdot \frac{\binom{k}{i}\binom{n-k}{k-i}}{\binom{n}{k}}. \tag{5.6}$$

Proof. Let $C_1, C_2 \in F$ be two clauses. Define the function $overlap(C_1, C_2) = |vars(C_1) \cap vars(C_2)|$. By definition, in order for the two clauses to be mergeable, they must both resolve and merge. First, the probability that C_1 and C_2 overlap on i variables is

$$Pr(overlap(C_1, C_2) = i) = \frac{\binom{k}{i}\binom{n-k}{k-i}}{\binom{n}{k}}.$$

Assume C_1 is fixed. The numerator denotes the number of ways k variables can be chosen for C_2, such that i variables overlap, and the remaining $(k-i)$ variables are chosen from variables not in C_1 (of which there are $n-k$). The denominator denotes all possible ways of choosing k variables from the full set of n.

If the two clauses overlap on i variables, then there are 2^i different ways in which the corresponding literals can either resolve or merge: simply fix the i literals in C_1, which leaves 2^i

k	n	m	Expected Merges
3	50	213	82.08
3	100	426	81.88
3	300	1280	81.99
5	60	1278	21293.78
5	80	1704	21650.82
5	100	2130	21862.04
7	30	2670	1112521.18
7	35	3115	1197603.51
7	40	3560	1262832.86

Table 5.1: Expected number of merges for various random-kSAT configurations.

possible polarity assignments of the literals in C_2. In order for the clauses to be mergeable, they must resolve on exactly one of the v variables (the remaining $i-1$ will merge):

$$Pr(resolves(C_1,C_2) \mid overlap(C_1,C_2)=i) = \frac{\binom{i}{1}}{2^i}.$$

Then the expected number of merges of the pair is:

$$\begin{aligned}
&E(mergesInPair(C_1,C_2)) \\
&= \sum_{i=1}^{k} (i-1) \cdot Pr(overlap(C_1,C_2)=i) \cdot Pr(resolves(C_1,C_2) \mid overlap(C_1,C_2)=i) \\
&= \sum_{i=1}^{k} (i-1) \cdot \frac{\binom{k}{i}\binom{n-k}{k-i}}{\binom{n}{k}} \cdot \frac{\binom{i}{1}}{2^i} \\
&= \sum_{i=1}^{k} \frac{i(i-1)}{2^i} \cdot \frac{\binom{k}{i}\binom{n-k}{k-i}}{\binom{n}{k}}
\end{aligned} \tag{5.7}$$

Since there are $\binom{m}{2}$ clause pairs in the formula, the expected number of merges in the formula is as stated in Equation 5.6. □

Table 5.1 displays the expected number of merges for varying k, n, and m at the empirical phase transition for each k. Interestingly, if we fix k and scale the size of the instance, the number of merges remains fairly constant. However, the number of merges increases dramatically as k is increased.

5.3 Experimental Setup

We first use the popularity-similarity random instance generator, which was introduced in [Giráldez-Cru and Levy, 2017], to create our base formulas. The generator has several parameters related to typical properties of industrial/application SAT instances. The *temperature* T allows one to tune the *similarity* in the instance: intuitively, during generation, each literal is assigned a random number, and if the temperature is low, literals with small differences in their random numbers are more likely to appear in the same clause. At higher temperatures, the generated formulas appear more close to traditional random SAT instances. We consider different temperature values between $1.8 - 100$, as listed in Table 5.2. We use the default *popularity* parameter $\beta = 1$, such that the variables are expected to occur according to a power-law distribution, as witnessed in industrial instances [Ansótegui et al., 2009]. We restrict the formulas to be 3SAT instances. The clause popularity parameter β' was set to 0, as in [Giráldez-Cru and Levy, 2017]. We always use a clause/variable ratio of 4.25. Since formulas with higher temperature are harder for CDCL solvers, formulas for $T < 2$ have 5000 variables, those with $2 \leq T < 2.2$ have 2000, those with $2.2 \leq T < 5$ have 1000, and those with $T \geq 5$ have 300. All other parameters are as default.

For each temperature value, we generate 10 base formulas for a total of 110 base formula. We then use our algorithm to increase or decrease the number of merges in each instance, creating a series of formulas associated with each base formula. Importantly, all considered formulas (both base formulas and those generated with our algorithm) are unsatisfiable.[2] We allot 1 hour for each run of our tool, and record the modified formula every 10 flips of literals, up to a limit of 500 flips (both for decreasing and increasing merges). Although each flip may cause many new merges, in our experiments, each flip tends to introduce one or two merges. In total, we considered 110 base formulas, and 1200 total formulas. We repeated this experiment for uniform random 3SAT instances at the phase transition (clause/variable ratio of 4.26), generating 100 base instances for each number of variables $n \in \{200, 225, 250\}$ (allowing both satisfiable and unsatisfiable instances in this case), and generating a new formula for every 5 flips of literals, in total producing 7043 formulas.

For each formula series (i.e., a base formula with its series of varied mergeability versions), we run MapleSAT as default with a 1 hour timeout and record solving time. We then compute the Spearman and Pearson correlations of mergeability versus solving time. Each row in Table 5.2 is aggregated over 10 formula series (correlations are aggregated using the Fisher transformation). Results are repeated for the uniform 3SAT instances in Table 5.3, split by satisfiability. More

[2] This naturally occurred for all temperature values $T < 5$, however a small number of satisfiable instances were generated for the largest T values. In order to maintain uniformity in this first experiment, these satisfiable instances were not included.

T	Spearman	Pearson	Min	Base	Max	Min Time (s)	Max Time (s)
T1.8	-0.9	-0.89	851.57	851.57	1155.86	1.23	0.6
T1.9	-0.92	-0.92	609	608.29	849.25	95.39	41.7
T2.0	-0.84	-0.89	332.6	363.89	508.9	8.53	5.89
T2.1	-0.92	-0.93	263	286.88	415.78	153.68	50.65
T2.2	-0.94	-0.93	95.4	224.9	326.7	6.88	2.63
T2.3	-0.97	-0.95	68.2	193.62	294.9	35.27	9.49
T2.4	-0.97	-0.94	70.3	183.22	261.7	126.07	18.31
T2.5	-0.98	-0.95	64.4	177.38	251.6	817.03	105.33
T5.0	-0.97	-0.98	42	99	137.4	12.98	4.94
T10.0	-0.97	-0.98	75.33	82	125	33.61	16.2
T100.0	-0.97	-0.95	69.43	82	119.14	94.43	35.24

Table 5.2: Correlations between mergeability and solving time for varying temperatures. Min (resp. Max) refer to the number of mergeable clause pairs in the formula with the least (resp. most) mergeable clause pairs in each formula series. Base is the number of mergeable pairs in the original instance not modified by our generator. Min Time and Max Time correspond to the times for the instance with the minimal (resp. maximal) number of mergeable pairs.

Num Vars	Spearman	Pearson	Min	Base	Max	Min Time (s)	Max Time (s)
200 (unsat)	-0.92	-0.92	52.84	81.6	123.04	2.48	0.74
225 (unsat)	-0.91	-0.91	65.46	83.46	123.11	5.84	2.61
250 (unsat)	-0.91	-0.91	56.6	84.73	123.97	13.53	6.38
200 (sat)	0.26	0.27	2.33	80.42	82.85	0.29	0.46
225 (sat)	0.27	0.30	18.99	81.76	92.76	0.97	1.74
250 (sat)	0.37	0.33	0.68	81.33	89.76	2.52	4.80

Table 5.3: Results for traditional random kSAT instances, split by satisfiability.

specifically, if a given series of instances has both satisfiable and unsatisfiable formulas, we distinguish each sub-series of instances with the same satisfiability.

5.4 Experimental Results

For the majority of formulas that are unsatisfiable, increasing the number of merges decreases solving time. The correlations, which are frequently greater in magnitude than 0.9, indicate a strong relationship over the benchmark. A possible explanation for this is that the additional

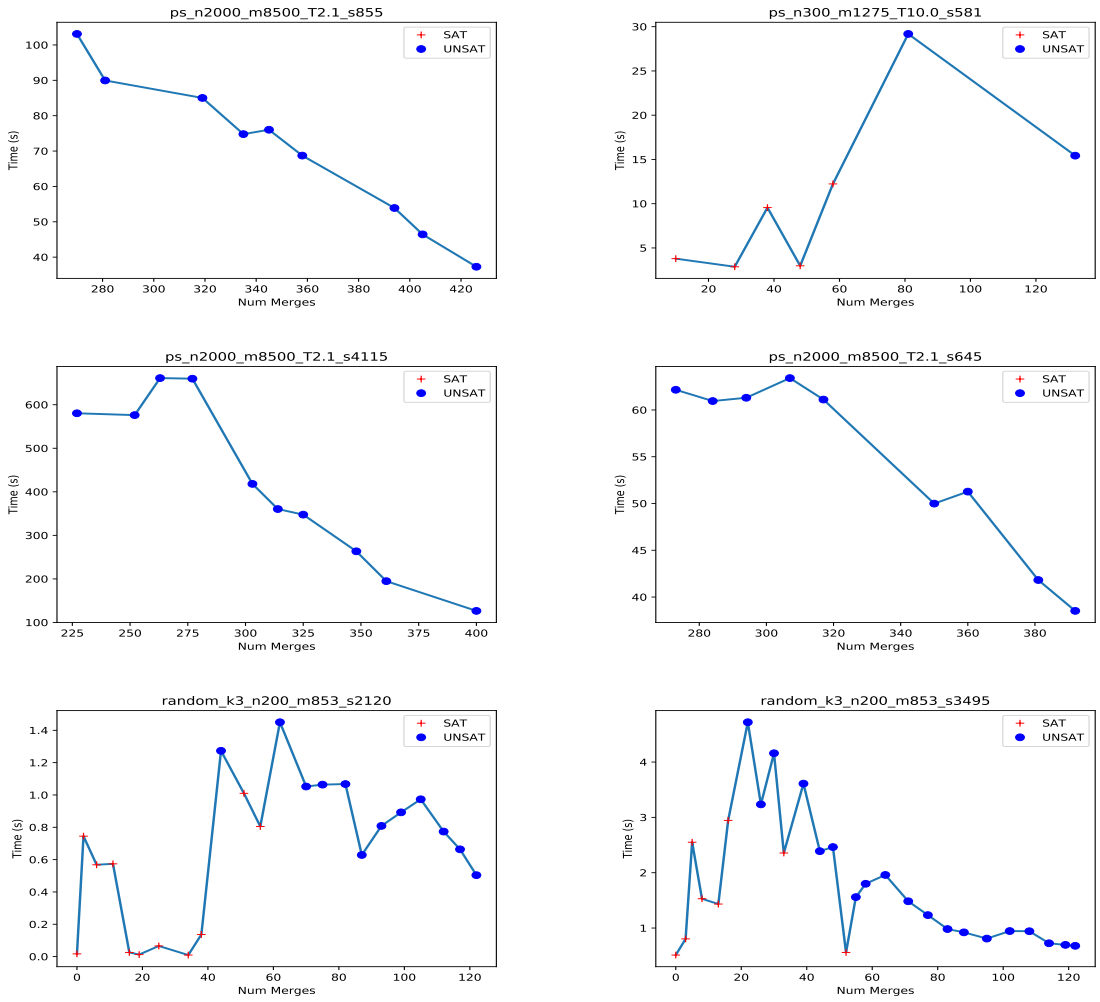


Figure 5.1: Scatter plots of several formula series. The number of merges is on each x-axis, and time in seconds is on the y-axis.

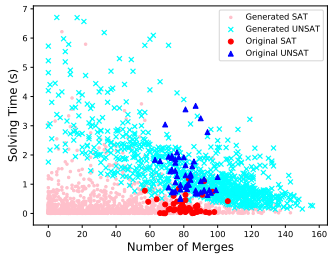
merges allow the solver to learn smaller clauses faster, restricting the search space more quickly. Interestingly, there is a slight positive correlation when considering satisfiable instances. Figure 5.1 depicts the relationship between the number of mergeable clause pairs and solving time for several series of formulas.

We further examined the set of uniform 3SAT instances with $n = 200$. In Figure 5.2a, we display the scatter plot of the number of mergeable clause pairs versus solving time, distinguishing instances based on satisfiability and how they were generated. From Equation 5.6, the expected number of mergeable clause pairs is 82. We computed the sample variance of the number of mergeable clause pairs using the original random 3SAT instances in order to compute the standard deviation over the sample, which was 9.32. In Figures 5.2b-d, we depict box plot distribution where the y-axis is again solving time, and the instances are grouped according to the number of standard deviations its merges are from the expected value of 82 merges, rounded toward zero. For example, an instance with 92 mergeable pairs would be 1.07 standard deviations from the expected value, and would fall into bucket 1. Note the clear downward slope over unsatisfiable instances in 5.2c. Also interestingly, the spread of solving times is much more significant if the number of mergeable pairs is small, as indicated by the many outliers on the left hand side of Figure 5.2b, whereas most instances are easily solvable when the number of mergeable pairs is high.

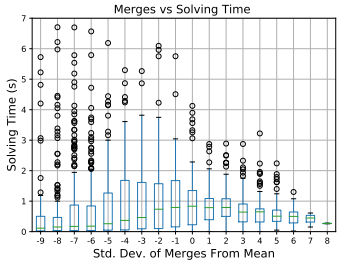
For satisfiable instances, there does not appear to be much correlation, however the Spearman correlations suggest a slight positive correlation. As one potential explanation for this, an increase in mergeability of satisfiable formulas may cause the formula to be more constrained by reducing the number of satisfying solutions to the formula (i.e., the #SAT value of the formula), making it in some sense "closer to unsatisfiable." We computed the #SAT value for each satisfiable formula, and computed the Spearman correlation for each formula series between #SAT and mergeability, which on average produced a -0.621 correlation coefficient. This suggests that increasing mergeability does often decrease the #SAT value of the formula.

Recall that an underlying motivation for studying mergeability was that the solver could learn shorter clauses during merge resolutions. A natural question is whether this occurs in practice. Figure 5.2e depicts a box plot comparing mergeability to the average learned clause size during search. As is apparent by the downward trend, instances that have more mergeable input tend to on average produce smaller learned clauses, supporting our intuition.

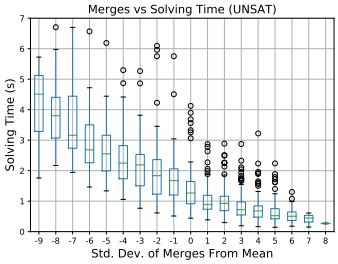
Last, although we intended to control for as many properties as possible when generating more mergeable formulas, we clearly cannot ensure that only the property of mergeability changes. A possible alternative explanation is that after increasing mergeability we introduce trivial unsatisfiable cores (i.e. a small subset of input clauses that are sufficient to derive UNSAT), offering an alternative explanation of the correlation. In Figure 5.2f, we empirically verify that



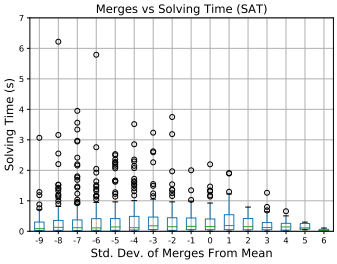
(a) Merges vs solving time.



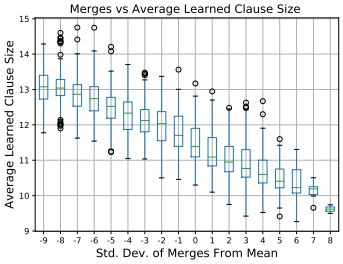
(b) All instances.



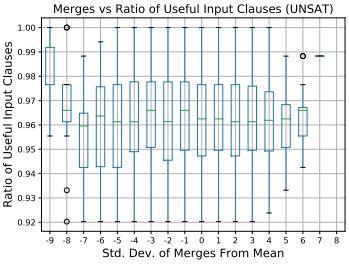
(c) Unsatisfiable instances.



(d) Satisfiable instances.



(e) Average learned clause size.



(f) Percentage of useful input clauses.

Figure 5.2: (a) Scatter plot depicting the distribution of 3SAT instances, comparing the number of merges of the input clauses on the x-axis, and solving time on the y-axis. Includes traditional random 3SAT instances at the phase transition with 200 variables, as well as scaled instances using our generator. (b)-(d) Box plot distribution where the y-axis is again solving time, and the instances are grouped according to the number of standard deviations (rounded toward zero) its merges are from the expected value of 82 merges. Note the clear downward slope over unsatisfiable instances in (c). (e) Average learned clause time tends to decrease as the number of mergeable pairs increases. (f) The percentage of useful input clauses is not affected by mergeability in our experiments.

this is not the case by measuring the number of *useful* input clauses for each SAT solver run of an unsatisfiable instance. We define a useful clause to an unsatisfiable proof as follows. Let P be the proof of unsatisfiability constructed by the SAT solver represented as a graph G, such that nodes represent clauses, input clauses have no incoming edges, and an edge exists from C_1 to C_2 *iff* the clause C_1 was in the implication graph used to derive C_2. (Additional edges are needed to account for extra components of real-world solvers, such as clause minimization.) The final node added to the graph is the empty clause E. Then, if we reverse all edges in the graph, the *useful clauses* correspond to the set of nodes reachable from E.

As can be seen by the graph, approximately 96% of the input clauses were useful to the proof, regardless of mergeability. While this does not necessarily calculate the minimal unsatisfiable core, it more closely reflects the actual run of the SAT solver. Thus, we do not believe that unsatisfiable core size is a confounding factor in determining the runtime of these instances.

Finally, in order to control for the possibility of our generator introducing additional unforseen changes to the formulas, we uniformly-randomly generated 10,000 3SAT instances at the phase transition. Among the unsatisfiable instances, we observed a negative Spearman correlation of -0.29, and for satisfiable instances a correlation of -0.02. While the correlation for unsatisfiable instances is not as strong in our experiments, the majority of these instances have mergeability close to the mean (sample standard deviation of mergeability was 8 for these instances), making it more difficult to see an apparent trend.

5.5 Conclusions

In order to isolate the effects of mergeability, we performed scaling studies, varying mergeability while keeping many other properties of the formula unchanged. Specifically, we describe a formula generator capable of scaling the mergeability parameter, and showed that mergeability of unsatisfiable instances tends to strongly correlate negatively with solver runtime (or strongly correlate positively with solver performance). We demonstrated that the expected number of mergeable input clauses pairs in uniform random kSAT instances is relative to the number of variables, clauses, and clause size. Further, the number of mergeable pairs increases significantly if the clause size in increased, while scaling up the number of variables and clauses, while fixing the clause size, does not increase the measure much. Also, we showed that the solver tends to produce shorter learned clauses on average for instances with higher mergeability. Given the strong correlations, we believe that mergeability may be a useful measure in portfolio solvers.

Chapter 6

LSR Backdoors to CDCL Solving

As our final contribution, we delve deeper into a new measure called *learning-sensitive with restarts (LSR) backdoors*, and demonstrate several separation results between various CDCL solving heuristics. Backdoors, in general, offer a natural way to characterize locality in Boolean formulas, but traditional backdoor definitions do not consider key features of CDCL solvers.

In [Dilkina et al., 2009b], the authors extended traditional backdoors to learning-sensitive (LS) backdoors in order to account for the power of clause-learning during the search performed by a SAT solver. They showed that LS backdoors are exponentially smaller than traditional strong backdoors on certain class of formulas. LSR backdoors naturally extend LS backdoors to allow restarts:

Definition 6 (Learning-sensitive with restarts (LSR) backdoor)**.** A set of variables $B \subseteq vars(F)$ is an LS backdoor with respect to a subsolver S if there exists a search tree exploration order with restarts, such that a clause-learning SAT solver branching only on variables in B, and with S as the subsolver at the leaves of the search tree, can determine the satisfiability of F.

In [Dilkina et al., 2009b], LS backdoors were only examined under a single configuration of a CDCL SAT solver, namely one that uses the first unique implication point (`1UIP`) learning scheme [Marques-Silva and Sakallah, 1999], and disallows restarts. Whereas our previous results were empirical in nature, we use LSR backdoors as a metric to compare various solver heuristics in theory, primarily by showing exponential separations in minimum backdoor sizes.

Main Contributions:

- In Section 6.2, we demonstrate that LSR backdoors are exponentially smaller than LS backdoors for certain class of instances, under the `1UIP` clause-learning scheme and when the solver is only allowed to backtrack instead of backjump.

- In Section 6.3, we show that the size of the LSR backdoor is dependent on the clause-learning scheme by demonstrating that LSR-`1UIP` backdoors may be exponentially smaller than LSR-`DL` backdoors (which use the decision learning (DL) scheme [Zhang et al., 2001]).

- In Section 6.4, we demonstrate several properties of LSR backdoors that are not apparent in traditional strong or weak backdoors. Specifically, we show that adding clauses to the formula may in some cases increase the LSR backdoor size. Further, we describe issues that may arise when a CDCL solver is attempting to witness an LSR backdoor (via some "perfect" branching sequence over the backdoor), if the solver must choose between multiple conflict clauses.

- In Section 6.5, we describe an algorithm, called *LaSeR*, to compute [overapproximations of] LSR backdoors using a single run of a CDCL solver. We further describe an approach to compute minimum sized LSR backdoors in Section 6.6.

- In Section 6.7, we show that rapid restart policies tend to produce more "local proofs" with respect to the variables in the computed LSR backdoor set, over a large set of industrial instances from SAT competitions. Informally, our notion of locality measures the set of unique variables in the "useful" learned clauses of the proof.

6.1 Base Formula Families

Dilkina et. al introduced two families of formulas that were used to demonstrate an exponential separation between LS backdoors and strong backdoors [Dilkina et al., 2009b]. We describe them here, as several variations of the formulas will be useful for deriving our results in the following sections.

We first introduce the following formula gadget, which will be useful to define the two formula families:

$$unit(q_\alpha, a_\alpha, b_\alpha) := (q_\alpha \vee a_\alpha) \wedge (q_\alpha \vee b_\alpha) \wedge (q_\alpha \vee \neg a_\alpha \vee \neg b_\alpha).$$

Note that any time we branch on the q_α variable and assign `false`, we will immediately reach a conflict using the three clauses. Clause learning will then derive the unit clause (q_α) (for both `1UIP` and `DL`).

Both formulas are defined on the variables $x_1, x_2, \ldots, x_n$, and also three auxiliary sets of variables $\{q_\alpha\}_{\alpha\in\{0,1\}^n}, \{a_\alpha\}_{\alpha\in\{0,1\}^n}, \{b_\alpha\}_{\alpha\in\{0,1\}^n}$, totalling $n + 3\cdot 2^n$ variables. For any assignment $\alpha \in \{0,1\}^n$ let $C_\alpha = x_1^{1-\alpha_1} \vee x_2^{1-\alpha_2} \vee \ldots \vee x_n^{1-\alpha_n}$ denote the clause on x variables which is uniquely

falsified by the assignment α, and let $X \mapsto \alpha$ mean that we branch on each x variable, such that $x_i = \alpha_i$. Throughout, we refer to the full set of x variables in the formula as X, and the full set of q variables as Q. The parameter $\mathscr{O}$ defines an ordering over the bit-strings in $\{0,1\}^n$. In [Dilkina et al., 2009b], this is assumed to be the lexicographic ordering, denoted LEX.

The Family $\mathscr{F}_{\mathscr{O}}$ of Boolean Formulas: Consider the formula

$$\mathscr{F}_{\mathscr{O}} = \bigwedge_{\alpha \in \{0,1\}^n} (C_\alpha \vee \bigvee_{\alpha' \leq_{\mathscr{O}} \alpha} \neg q_{\alpha'}) \wedge unit(q_\alpha, a_\alpha, b_\alpha) \tag{6.1}$$

Example 3. We state the set of clauses in $\mathscr{F}_{LEX}$ with $n = 3$, which we use throughout in future examples:

$$\begin{aligned}
&(x_0 \vee x_1 \vee x_2 \vee \neg q_{000}) \wedge \\
&(x_0 \vee x_1 \vee \neg x_2 \vee \neg q_{000} \vee \neg q_{001}) \wedge \\
&(x_0 \vee \neg x_1 \vee x_2 \vee \neg q_{000} \vee \neg q_{001} \vee \neg q_{010}) \wedge \\
&(x_0 \vee \neg x_1 \vee \neg x_2 \vee \neg q_{000} \vee \neg q_{001} \vee \neg q_{010} \vee \neg q_{011}) \wedge \\
&(\neg x_0 \vee x_1 \vee x_2 \vee \neg q_{000} \vee \neg q_{001} \vee \neg q_{010} \vee \neg q_{011} \vee \neg q_{100}) \wedge \\
&(\neg x_0 \vee x_1 \vee \neg x_2 \vee \neg q_{000} \vee \neg q_{001} \vee \neg q_{010} \vee \neg q_{011} \vee \neg q_{100} \vee \neg q_{101}) \wedge \\
&(\neg x_0 \vee \neg x_1 \vee x_2 \vee \neg q_{000} \vee \neg q_{001} \vee \neg q_{010} \vee \neg q_{011} \vee \neg q_{100} \vee \neg q_{101} \vee \neg q_{110}) \wedge \\
&(\neg x_0 \vee \neg x_1 \vee \neg x_2 \vee \neg q_{000} \vee \neg q_{001} \vee \neg q_{010} \vee \neg q_{011} \vee \neg q_{100} \vee \neg q_{101} \vee \neg q_{110} \vee \neg q_{111}) \wedge \\
&(q_{000} \vee a_{000}) \wedge (q_{000} \vee b_{000}) \wedge (q_{000} \vee \neg a_{000} \vee \neg b_{000}) \wedge \\
&(q_{001} \vee a_{001}) \wedge (q_{001} \vee b_{001}) \wedge (q_{001} \vee \neg a_{001} \vee \neg b_{001}) \wedge \\
&\ldots \\
&(q_{111} \vee a_{111}) \wedge (q_{111} \vee b_{111}) \wedge (q_{111} \vee \neg a_{111} \vee \neg b_{111})
\end{aligned} \tag{6.2}$$

The smallest LS backdoor for F_{LEX} is the set of x variables and therefore of size n, which can be shown as follows. To see that X is an LS backdoor, we first branch on all X variables setting them to `false`. This then propagates $\neg q_{0\ldots0}$, which leads to learning $(q_{0\ldots0})$, as above. Since the learned clause is unit, the solver backjumps to decision level 0, at which point we repeat the process by assigning X according to the next lexicographic assignment $0\ldots01$ (i.e., we set $x_1, \ldots, x_{n-1}$ to `false`, and x_n to `true`). Repeating this, we derive all (q_α) clauses in lexicographic order and eventually derive UNSAT. Note importantly that when deriving each (q_α), all previously learned clauses $\{(q_{\alpha'}) \mid \alpha' \leq \alpha\}$ are used for propagation, so branching in lexicographic order is essential.

The Family $\mathscr{G}_{\mathscr{O}}$ of Boolean Formulas: Now, consider the formula

$$\mathscr{G}_{\mathscr{O}} = \bigwedge_{\substack{\alpha \in \{0,1\}^n, \\ \alpha \neq 1^n}} \big((C_\alpha \vee \neg q_\alpha) \wedge \mathit{unit}(q_\alpha, a_\alpha, b_\alpha)\big) \wedge (C_{1^n} \vee \bigvee_{\alpha'} \neg q_{\alpha'}) \wedge \mathit{unit}(q_{1^n}, a_{1^n}, b_{1^n}) \tag{6.3}$$

The smallest LS backdoor for $\mathscr{G}_{LEX}$ is again the X variables and follows similarly to $\mathscr{F}_{LEX}$. We iterate through all assignments to X variables in lexicographic order, learning each unit clause (q_α) along the way. However, the final assignment 1^n is treated differently, and importantly must be queried last. After learning all previous (q_α) clauses and assigning the X variables according to 1^n, we propagate $\neg q_{1^n}$, which again leads to conflict in the usual manner, ultimating proving the formula to be UNSAT.

6.2 Separating LSR Backdoors from LS Backdoors

In this section we prove that for certain classes of formulas, the minimal LSR backdoors are exponentially smaller than the minimal LS backdoors under the assumption that the learning scheme is `1UIP` and that the CDCL solver is only allowed to backtrack (and not backjump). Backjumping raises several issues, particularly when unit clauses are learned. Upon learning a clause, CDCL solvers with backjumping return to the second highest decision level in the learned clause (which defaults to level 0 for learned unit clauses). Consequently, backjumping after learning a clause effectively allow these solvers to get a "free" restart. Hence, we do not find our assumption to be too unreasonable.

The formula $F_{\mathscr{O}}$ was introduced in [Dilkina et al., 2009b], however they only used the lexicographic ordering which enabled them to demonstrate a separation between LS backdoors and strong backdoors. The key insight was that if a CDCL solver without restarts queried the $x_1, \ldots, x_n$ variables in lexicographic ordering of assignments, it would learn crucial conflict clauses that would enable the solver to establish the unsatisfiability of the instance without having to query any additional variables. (By the term "querying a variable" we mean that the solver decides a value to it and propagates it.) Since strong backdoors are defined in a way that they cannot benefit from clause learning they would necessarily have to query additional variables.

LSR Backdoors for $\mathscr{F}_{\mathscr{O}}$ Formulas: In the lemma below, we show that the X variables in the formula family $\mathscr{F}_{\mathscr{O}}$ constitute an LSR backdoor.

Lemma 1. Let $\mathscr{O}$ be any ordering of $\{0,1\}^n$. The X variables form a LSR backdoor for formulas in the family $\mathscr{F}_{\mathscr{O}}$.

Proof. Query the x variables according to the ordering given by $\mathscr{O}$. As soon as we have a complete assignment to the x variables, we will unit-propagate to a conflict and learn a q_α variable as a conflict clause; after that we restart. Once all such assignments are explored we can simply query the X variables in any order (without restarts) to yield a contradiction, since every assignment to the X variables will falsify the formula. □

Lower Bound on the Size of LS Backdoors for $\mathscr{F}_\mathscr{O}$ Formulas: We know $\mathscr{F}_\mathscr{O}$ is minimally unsatisfiable from [Dilkina et al., 2009b] (this is regardless of the ordering $\mathscr{O}$). First, let us define the following subsets of clauses. Let $X_\mathscr{O}$ be the 2^n clauses of $\mathscr{F}_\mathscr{O}$ that contain some C_α disjunction, and for every α, Q_α be the three clauses over $\{q_\alpha, a_\alpha, b_\alpha\}$ that allow us to learn each q_α (note that these Q_α clauses are never affected by the ordering $\mathscr{O}$). We start with the following observation:

Lemma 2. For any ordering $\mathscr{O}$, $x \in X$, and polarity b, let $\mathscr{F}_\mathscr{O}[x=b]$ be the residual formula after setting $x=b$. Let C be any of the remaining clauses from $X_\mathscr{O}$. Then $\mathscr{F}_\mathscr{O}[x=b] \setminus \{C\}$ is satisfiable.

Example 4. The following is the residual formula for $\mathscr{F}_{LEX}[x_2 = T]$ for $n=3$:

$$
\begin{aligned}
&(x_0 \vee x_1 \vee \neg q_{000} \vee \neg q_{001}) \wedge \\
&(x_0 \vee \neg x_1 \vee \neg q_{000} \vee \neg q_{001} \vee \neg q_{010} \vee \neg q_{011}) \wedge \\
&(\neg x_0 \vee x_1 \vee \neg q_{000} \vee \neg q_{001} \vee \neg q_{010} \vee \neg q_{011} \vee \neg q_{100} \vee \neg q_{101}) \wedge \\
&(\neg x_0 \vee \neg x_1 \vee \neg q_{000} \vee \neg q_{001} \vee \neg q_{010} \vee \neg q_{011} \vee \neg q_{100} \vee \neg q_{101} \vee \neg q_{110} \vee \neg q_{111}) \wedge \\
&(q_{000} \vee a_{000}) \wedge (q_{000} \vee b_{000}) \wedge (q_{000} \vee \neg a_{000} \vee \neg b_{000}) \wedge \\
&(q_{001} \vee a_{001}) \wedge (q_{001} \vee b_{001}) \wedge (q_{001} \vee \neg a_{001} \vee \neg b_{001}) \wedge \\
&\ldots \\
&(q_{111} \vee a_{111}) \wedge (q_{111} \vee b_{111}) \wedge (q_{111} \vee \neg a_{111} \vee \neg b_{111})
\end{aligned}
\tag{6.4}
$$

Proof. Suppose we remove any clause $C \in X_\mathscr{O}[x=b]$. Then we can satisfy the remaining clauses by setting $q_\alpha = T$ for every α, and $X \setminus \{x\}$ according to the opposite polarity of the literals in the C. □

Importantly, any proof of unsatisfiability of $\mathscr{F}_\mathscr{O}[x=b]$ must therefore "make use of" all remaining clauses in $X_\mathscr{O}$.[1] We first consider the case where some $x=b$ is the first decision the solver makes (as opposed to assigning some q, a, or b variable). For a clause $C \in X_\mathscr{O}$, let $numQ(C)$ be the number of q literals in C, and for two clauses $C, C_2 \in X_\mathscr{O}$, let $diffQ(C, C_2) =$

[1] More precisely, any unsatisfiable core of $\mathscr{F}_\mathscr{O}[x=b]$ must contain the clauses of $X_\mathscr{O}[x=b]$.

$|numQ(C) - numQ(C_2)| - 1$, be the difference in the number of q literals between the two clauses, minus 1. Note that by the construction of $\mathscr{F}$, the set of q literals in the smaller clause will always be a subset of the larger clause.

Lemma 3. For any $\mathscr{F}_{\mathscr{O}}[x = b]$, let B be a minimum LS backdoor (even with restarts). Define the sequence σ over the clauses of $X_{\mathscr{O}}[x = b]$ such that for every i, $numQ(\sigma_i) < numQ(\sigma_{i+1})$. Then

$$|B| \geq (\text{numQ}(\sigma_1) - 1) + \sum_{i=1}^{|\sigma|-1} \text{diffQ}(\sigma_i, \sigma_{i+1}). \tag{6.5}$$

Example 5. Consider again the residual formula in Example 6.4. The sequence σ is defined over the first four clauses in the listed order. The respective value for *numQ()* for each of the four clauses in order is 2, 4, 6, 8. Equation 6.5 intuitively counts the number of skipped values in this sequence between 1 and the highest number: $|B| \geq 8 - |\{1,3,5,7\}| = 4$.

Proof. We first note that there are three ways in which some q_α can be assigned. First, we can simply query q_α. Second, we can query a_α and b_α to propagate q_α; since a_α and b_α are "local" to the three clauses of Q_α, we always choose to branch on the single q_α variable instead of the two a_α and b_α variables, which would only increase our backdoor size. Thus, we ignore this case and only ever branch on q and x variables. Third, for some clause in $X_{\mathscr{O}}[x = b]$ which includes q_α, we can falsify all literals except $\neg q_\alpha$ which will propagate it.

As stated in Lemma 2, any proof of unsatisfiability of $\mathscr{F}_{\mathscr{O}}[x = b]$ must make use of all remaining clauses in $X_{\mathscr{O}}$, specifically by propagation of some variable in the clause. We show by induction on the length of any prefix of σ that Equation 6.5 must hold. Suppose we wish to propagate on σ_1. Clearly, propagating first on any later σ_i (or q variables not present in σ_1) will only require more queries of q variables. So, we must branch on all but one variable in σ_1 in order to propagate on it, which in total is $n + numQ(\sigma_1) - 1$, where n is again the number of x variables, so the base case holds.

Suppose our prefix of σ has length m, and that in order to propagate on the first $m-1$ clauses, we needed to branch on

$$B \geq (\text{numQ}(\sigma_1) - 1) + \sum_{i=1}^{m-2} \text{diffQ}(\sigma_i, \sigma_{i+1}) \tag{6.6}$$

Since σ_m has strictly more q literals than the previous $m-1$ clauses by construction, we could not have branched upon new q literals in σ_m earlier to decrease our backdoor set. To see that we must additionally branch on $diffQ(\sigma_{m-1}, \sigma_m)$ new variables, again note that the q literals in σ_m are a strict superset of any previous σ_i, and therefore could not have been propagated by any previous σ_i. Thus, we must query at least $diffQ(\sigma_{m-1}, \sigma_m)$ new q literals in order to propagate on σ_m. □

Note that the above proof assumed that we first branched on an x variable (i.e. by setting $x = b$); we now argue that we can branch on some x variable first without loss of generality. We have already argued that we should not branch on any a_α or b_α, and therefore $B \subseteq X \cup Q$, so suppose we branched on some q literals first. Then, without loss of generality, we can assume that when q_α is queried, it is set to false. To see this, notice that querying $q_\alpha = \texttt{false}$ will immediately unit propagate to a conflict, and 1UIP-learning will immediately yield the unit clause q_α. Thus we can always replace queries of the form $q_\alpha = \texttt{true}$ *in-situ* with queries $q_\alpha = \texttt{false}$ without affecting the rest of the algorithm's execution. Suppose we branch on an arbitrary number of q variables before branching on some x variable. After this sequence of decisions, we will have learned a unit clause (q) for every queried q variable, and x will be on the trail. Since our solver performs backtracking instead of backjumping, we can derive the same state of the solver by first querying x, followed by the q variables.[2] Thus, we can assume that we branch on some x variable first.

Importantly, once a non-backjumping solver without restarts assigns any x variable to b, Lemma 3 can be applied, since the solver must falsify the entire $x = b$ branch before solving the remainder of the formula. Thus, if we find an ordering $\mathscr{O}$ such that for any x and b, the LS backdoor of $\mathscr{F}_\mathscr{O}[x = b]$ has size exponential in n, then the LS backdoor of $\mathscr{F}_\mathscr{O}$ must be at least as large.

Recall that a *decision tree* is a binary tree where nodes are labelled with variables and the two outgoing edges from a node denote polarity assignents to the variable. We encode the execution of the CDCL algorithm as a decision tree, denoting the order in which we query all complete assignments to the X variables. Note that we must assign all X variables to hit a conflict by the structure of $\mathscr{F}_\mathscr{O}$, and so we let T denote the complete depth-n decision tree querying the X variables obtained from the CDCL execution tree.

Key Property. For any decision tree T there is a coordinate $i \in [n]$ and $b \in \{0,1\}$ such that for bit-strings $\alpha_j, \alpha_k \in \{0,1\}^n$,

$$\alpha_j[i] = b, \alpha_k[i] = 1 - b$$

for all $j = 1, 2, \ldots, 2^n/2$ and $k = 2^n/2 + 1, 2^n/2 + 2, \ldots, 2^n$.

To see this, simply let i be the index of the variable labelled on the root of the decision tree T. If the decision tree queries the bit b in the left subtree then the first half of the strings in the ordering will have the ith bit of the string set to b, and the second half of strings set to $1 - b$.

[2] If our solver was allowed to backjump, then after querying x followed by some $\neg q$, since we would learn the unit clause (q), the solver would backjump to decision level 0, clearing x from the trail.

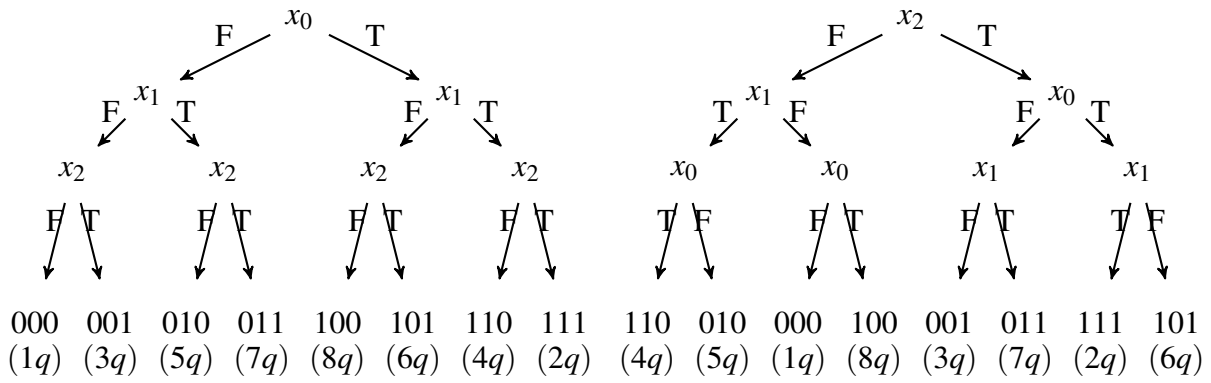


Figure 6.1: Two example decision trees for the formula $\mathscr{F}_{SCR}$ with $n = 3$.

To use this property, let $\beta_1, \beta_2, \ldots, \beta_{2^n}$ be the lexicographic ordering of $\{0,1\}^n$, and for any string β_i define $\overline{\beta_i}$ to be the string obtained by flipping each bit in β_i. Then define the scrambled ordering *SCR* as

$$\beta_1, \overline{\beta}_1, \beta_2, \overline{\beta}_2, \ldots, \beta_{2^n/2}, \overline{\beta}_{2^n/2}. \tag{6.7}$$

It follows that for *any* x and b, half of the bit-strings $\alpha \in \{0,1\}^n$ with $\alpha[i] = b$ will be in the first half of the ordering *SCR*, and the other half with $\alpha[i] = b$ are in the second half of the ordering.

Example 6. Figure 6.1 depicts two example decision trees for the formula $\mathscr{F}_{SCR}$ with $n = 3$. Each tree should be interpreted as the order in which the CDCL solver considers each assignment to the X variables, read left to right. Leaves correspond to the bit-strings of the X variable assignments, and the parenthesized q values beneath represent how many q variables are in the associated clause in X_{SCR}. For example, the clause associated with 001 is $(x_0 \vee x_1 \vee \neg x_2 \vee \neg q_{000} \vee \neg q_{001} \vee \neg q_{010})$.

Lemma 4. For $n > 3$ the smallest LS backdoor for the formula $\mathscr{F}_{SCR}$ has size at least $2^{n-2} - 3$.

Proof. Consider the following key clauses: the clause associated with $\beta_{2^n/2}$ (which has $2^n - 1$ q literals, and the one associated with $\overline{\beta}_{2^n/2}$ (which has 2^n q literals). We call these the *long* clauses. Further the two *short* clauses are associated with β_1 and $\overline{\beta}_1$, which respectively have one q literal and two q literals. As argued above, assume we first assign some x variable to b. Then regardless of choice of x and b, one of the long clauses will remain in $X_{\mathscr{O}}[x = b]$, and so will one of the short clauses.

Let the ordering σ of the clauses in X_{SCR} be defined as in Lemma 3. Through a simple counting argument, since there are only 2^{n-1} clauses in $X_{SCR}[x=b]$, $numQ(\sigma_1) \leq 2$ (i.e. from a short clause), and $numQ(\sigma_{2^{n-1}}) \geq 2^n - 1$ (i.e. from a long clause), the summation of Equation 6.5 of Lemma 3 must equal at least $2^{n-2} - 3$. □

Theorem 3. For every $n > 3$, there is a formula on $N = O(2^n)$ variables such that the minimal LSR backdoor has $O(\log N)$ variables, but every LS backdoor has $\Omega(N)$ variables.

Proof. Consider $\mathscr{F}_{\mathscr{O}}$ with the ordering *SCR* as described in Lemma 4. We know that the smallest LS backdoor for $\mathscr{F}_{SCR}$ has at least $2^{n-2} - 3$ variables. By Lemma 1, the X variables constitute an LSR backdoor of size n. □

6.3 The Effect of Clause-Learning Schemes

We next show that the size of the minimal LSR backdoor is dependent on the solver's underlying clause-learning scheme. We draw comparisons between the `1UIP` and `DL` schemes. Note that for all following results, we allow backjumping (as opposed to just backtracking as in the previous subsection). A takeaway from these results is that the LSR backdoor gives us a deeper theoretical understanding of why the `1UIP` learning scheme can remain more local than the `DL` learning scheme.

Before describing our results, we first recall the properties of *absorption*, *1-empowerment*, and *1-provability*, which were initially used to demonstrate that CDCL can simulate general resolution within some polynomial-size bound:

Definition 7 (Absorption [Atserias et al., 2011])**.** Let Δ be a set of clauses, let C be a non-empty clause and let x^α be a literal in C. Then Δ absorbs C at x^α if every non-conflicting state of the solver that falsifies $C \setminus \{x^\alpha\}$ assigns x to α. If Δ absorbs C at every literal, then Δ absorbs C.

The intuition behind absorbed clauses is that adding an already absorbed clause C to Δ is in some sense redundant, since any unit propagation that could have been realized with C is already realized by clauses in Δ.

Definition 8 (1-Empowerment [Pipatsrisawat and Darwiche, 2008])**.** Let $\alpha \Rightarrow l$ be a clause where l is some literal in the clause and α is a conjunction of literals. The clause is 1-empowering with respect to a set of clauses Δ if:

1. $\Delta \models (\alpha \Rightarrow l)$: the clause is implied by Δ.

2. $\Delta \wedge \alpha$ does not result in a conflict detectable by unit propagation.

3. $\Delta \wedge \alpha \not\vdash_1 l$: unit propagation cannot derive l after asserting the literals in α.

Definition 9 (1-Provability [Pipatsrisawat and Darwiche, 2009])**.** Given a set of clauses Δ, a clause C is 1-provable with respect to Δ *iff* $\Delta \wedge \neg C \vdash_1 false$.

An important note is that every learned clause is both 1-empowering and 1-provable, and therefore not absorbed, at the moment it is derived by a CDCL solver (i.e., before being added to Δ) [Pipatsrisawat and Darwiche, 2008, Pipatsrisawat and Darwiche, 2009].

Lemma 5. Let Δ be a set of clauses and suppose that C is a 1-empowering and 1-provable clause with respect to Δ. Then there exists a sequence σ of decisions and restarts containing only variables from C such that Δ and the set of learned clauses obtained from applying σ absorbs C.

Proof. The proof follows directly from the construction of such a decision sequence in the proof of Proposition 2 of [Pipatsrisawat and Darwiche, 2009]. □

Theorem 4. Let F be a formula with an LSR-DL backdoor of size n. Then the smallest LSR-1UIP backdoor for F has size at most n.

Proof. Consider the sequence of learned clauses in the proof that witnesses the smallest LSR-DL backdoor. Then the DL-solver must have branched on all the variables in the learned clauses given the nature of the DL scheme. Let B be this set of variables. Then we can absorb the clauses in the sequence one-by-one by only branching on the variables in those clauses. □

Theorem 5. There exists an infinite family of formulas such that the smallest LSR-DL (resp. LS-DL) backdoor for each instance is exponentially larger than the smallest LSR-1UIP (resp. LS-1UIP) backdoor.

Proof. We show this using the formula family $\mathscr{F}_{LEX}$. The result follows analogously to the separation of LS-1UIP backdoors and strong backdoors in [Dilkina et al., 2009b]. We have already demonstrated that the smallest LSR-1UIP backdoor is of size $|X| = n$ (this is also the case for LS-1UIP backdoors [Dilkina et al., 2009a]).

Since each formula in $\mathscr{F}_{LEX}$ is minimally unsatisfiable, in order to derive UNSAT we must "make use" of each clause through some propagation. Let C_{long} be the clause with the largest number of q_α literals. If our branching sequence only branched previously on variables in X, then all learned clauses will only include variables in X, and in particular could not propagate the Q variables in C_{long}. The only way we could have derived (q_α) clauses previously is through branching on them, which would also increase the size of the backdoor. Thus, we must branch on at least the n variables from X and $2^n - 1$ variables from Q in order to propagate on C_{long}. □

6.4 Further Properties of LS and LSR Backdoors

In the case of traditional strong or weak backdoors with UP as the subsolver, for a given formula F, it is easy to show that adding clauses to F can only decrease the size of the backdoor. This is not the case for LS and LSR backdoors.

Observation 6. Given formulas F_1 and F_2, the formula $F_1 \wedge F_2$ may have a larger LSR (or LS) backdoor than either individual formula.

Example 7. Consider $F_1 \in \mathscr{F}_{LEX}$, and let F_2 be the single unit clause (x_1). Note that F_2 subsumes the first half of the clauses defined over C_α's (since we are using the lexicographic ordering). Therefore, we can no longer utilize those clauses to derive conflicts, since the solver will never set x_1 to false. Through the same argument as in Lemma 3, it is easy to show that the solver must begin branching on variables not in X in order to solve the formula.

Next, we show that even if the solver is given a perfect branching sequence witnessing an LSR backdoor, there exist formulas where the solver may still need to branch on additional variables. Our result relies on the following probabilistic assumption:

Definition 10 (Uniform Conflict Choice (UCC) Assumption)**.** Let Δ be a set of clauses and D be a sequence of decisions, such that for any proper prefix P of D, $\Delta \wedge P$ is 1-consistent, but $\Delta \wedge D$ is 1-inconsistent, i.e., it causes a conflict. Further, assume that there are n unique conflict clauses that can be derived after branching on D, depending on the order in which literals are propagated. The Uniform Conflict Choice assumption states that the solver always chooses the conflict clause to learn uniformly at random from the family of possible conflict clauses.

Example 8. Consider the implication graph in Figure 6.2. Depending on whether q or r is propagated first, a solver using `1UIP` may learn the unit clause $(\neg q)$ or $(\neg r)$, respectively. Under the UCC assumption, each has a 50% likelihood of being derived. Since solvers typically only learn one clause per conflict, after one of the two clauses is learned, the solver will backjump to decision level 0 (since both clauses are unit), ignoring the other possible clause.

Theorem 6. There exists an infinite family of formulas such that for any $\delta > 0$, the probability of realizing the minimal LS backdoor (or LSR backdoor), even given the perfect branching sequence, is less than δ, under the UCC assumption.

Proof. We construct a new family of formulas $\mathscr{G}_{2LEX}$ by starting with $\mathscr{G}_{LEX}$ and adding a duplicate set of clauses, where the X variables are reused in these clauses, but each q_α, a_α, and b_α is replaced

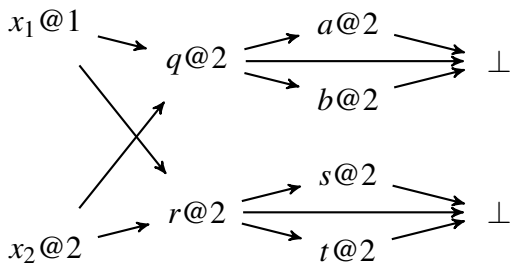


Figure 6.2: Example of multiple conflicts after making decisions x_1 and x_2. The number after the '@' symbol denotes the decision level of the literal.

by a fresh r_α, s_α, and t_α, respectively:

$$\begin{aligned}\mathscr{G}_{2\mathscr{O}} = & \bigwedge_{\alpha\in\{0,1\}^n,\alpha\neq 1^n} \big((C_\alpha \vee \neg q_\alpha) \wedge unit(q_\alpha, a_\alpha, b_\alpha)\big) \wedge \\ & \bigwedge_{\alpha\in\{0,1\}^n,\alpha\neq 1^n} \big((C_\alpha \vee \neg r_\alpha) \wedge unit(r_\alpha, s_\alpha, t_\alpha)\big) \wedge \\ & (C_{1^n} \vee \bigvee_{\alpha'} \neg q_{\alpha'}) \wedge unit(q_{1^n}, a_{1^n}, b_{1^n}) \wedge \\ & (C_{1^n} \vee \bigvee_{\alpha'} \neg r_{\alpha'}) \wedge unit(r_{1^n}, s_{1^n}, t_{1^n}).\end{aligned} \tag{6.8}$$

Note that the first and third lines are exactly the clauses from $\mathscr{G}_{\mathscr{O}}$. We first show that, as in the case for $\mathscr{G}_{LEX}$, the X variables constitute an LS backdoor for $\mathscr{G}_{2LEX}$. Let P be the branching sequence that witnesses X as an LS backdoor for $\mathscr{G}_{LEX}$ (i.e. by branching on X lexicographically). For each assignment α to X ($\alpha \neq 1^n$), the solver can derive one of two conflict clauses depending on the order of propagations: either (q_α) or (r_α) (as in Example 6.2). If for every $\alpha \neq 1^n$ the solver derives some q_α, then the solver can derive the same proof as derived for the $\mathscr{G}_{LEX}$ formula. This effectively ignores any clauses that contain some r_α. The same holds if r_α is always chosen, and the clauses with q_α are ignored. Thus, the branching sequence P witnesses that X is an LS backdoor for $\mathscr{G}_{2LEX}$. Further, it is clear that branching on any q, r, a, b, s, or t variables cannot reduce the size of the backdoor through a similar argument as in the case for $\mathscr{G}_{LEX}$, and also given the fact that no q or r variables share a clause. Thus X is the smallest backdoor for $\mathscr{G}_{2LEX}$.

We next show that for any α, if the solver learns some unit clause (q_α), then it can never learn (r_α), unless it queries the variable r_α (similarly q_α must be queried if (r_α) is first learned).

Suppose *w.l.o.g.* that we learn the clause (q_α). Then the only clause that can be used to propagate $\neg r_\alpha$ is $(C_\alpha \vee \neg r_\alpha)$. However, upon assigning $n-1$ of the X variables in C_α, because we now know (q_α), we will propagate the final X variable in such a way that C_α is satisfied. Therefore, we cannot use this clause to propagate r_α, and we must query r_α in order to assign it.[3]

Now suppose that instead of either always learning q_α, or always learning r_α, that a mix of the two are learned. Then, when the final lexicographic assignment is reached, which sets $X \mapsto 1 \dots 1$, we are not able to propagate the final literal (either q_{1^n} or r_{1^n}), since the clauses listed on lines 3 and 4 of Equation 6.8 will have multiple unassigned literals. Thus, the solver is forced to branch on q or r literals to derive UNSAT.

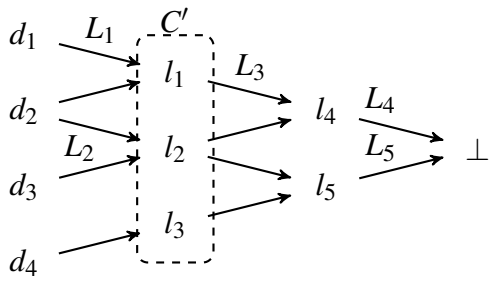
It remains to compute the probability of this occurring under the UCC assumption. Given $|X| = n$, there are $2^n - 1$ assignments to X that occur before the conflict involving C_{1^n}, and for each assignment α, we can learn either (q_α) or (r_α). Then the likelihood of picking either all q_α's or all r_α's is $2/(2^n - 1)$. Given a fixed δ, choosing any $n \geq \lceil log_2(2/\delta) \rceil + 1$ completes our result. □

We note that the above result will not hold for the $\mathscr{F}_{LEX}$ family of formulas, due to the interdependencies between the clauses that contain some C_α. For example, suppose we performed an analogous duplication of clauses in $\mathscr{F}_{LEX}$ to create the formulas $\mathscr{F}_{2LEX}$ (again not duplicating the X variables). Consider the lexicographic branching sequence that witnesses X as an LS backdoor for $\mathscr{F}_{LEX}$, which also witnesses X as an LS backdoor for $\mathscr{F}_{2LEX}$. After branching on the X variables the first time, in $\mathscr{F}_{2LEX}$, we can again either learn $q_{0\dots0}$ or $r_{0\dots0}$. Suppose w.l.o.g. that we learn $q_{0\dots0}$. Then in the second iteration where we assign $X \mapsto 0 \dots 01$, because we have already learned $q_{0\dots0}$, unit propagation will assign and learn a conflict over $\neg q_{0\dots1}$, and will not have the option to learn a clause over $r_{0\dots1}$. Thus, all future choices between learning q_α or r_α are determined by the first choice, and the perfect branching sequence will always succeed.

Corollary 1. There exists an infinite family of formulas such for any $\delta > 1$, the expected size of the LS backdoor is at least $\delta \cdot |B|$, where $|B|$ is the size of the minimal backdoor, even with perfect branching, under the UCC assumption.

Proof. The result follows from Theorem 6. □

[3] Note again that we ignore the cases of assigning s or t variables as this would only increase the backdoor size. We could also propagate r_α on the long clause containing all 2^n r variables, however this would significantly increase the backdoor size as well.



$$C' = (\neg l_1 \vee \neg l_2 \vee \neg l_3)$$
$$R^*_{L_3} = \{L_6, L_7\}$$
$$R^*_{L_4} = \{\}$$
$$R^*_{L_5} = \{L_6\}$$

$$R_{C'} = \{L_3, L_4, L_5\}$$
$$R^*_{C'} = \{L_3, L_4, L_5, L_6, L_7\}$$
$$D^*_{C'} = \{var(l_1), var(l_2), var(l_3)\} \cup vars(\{L_3, L_4, L_5, L_6, L_7\})$$

Figure 6.3: Example conflict analysis graph depicting the set of relevant clauses and variables to some learned clause C'. Nodes are literals. Edges labeled with some L_i are previously learned clauses; all other edges depicting propagations are from the original formula F. The clauses L_6, L_7 used to derive L_3 and L_5 are not shown, but would be in the respective conflict analysis graphs of L_3 and L_5. The clauses L_1 and L_2 are not included in $R_{C'}$ since they occur on the reason side of the graph.

6.5 Computing LSR Backdoors through Absorption

Dilkina et al. [Dilkina et al., 2009a, Dilkina et al., 2014] described an approach for empirically computing upper bounds on minimal LS backdoors. In this section, we explore a novel connection between backdoors and the notion of clause absorption, which allows us to develop a new approach to computing LSR backdoors.

We propose a new approach that takes advantage of allowing restarts which can often greatly reduce the number of decisions necessary to construct such a backdoor (as compared to [Dilkina et al., 2014]), especially if many "unnecessary" clauses are derived during solving. Our key insight is that, as stated in [Ansótegui et al., 2015, Oh, 2015], most learned clauses are ultimately not used to determine the satisfiability of a formula, and therefore we only need to consider variables required to derive such "useful" clauses. Our result shows that, for an unsatisfiable formula, the set of variables within the set of learned clauses in the UNSAT proof constitutes an LSR backdoor. The result for satisfiable formulas shows that the set of decision variables in the final trail of the solver, along with the variables in certain learned clauses, constitute an LSR backdoor.

Our result additionally makes use of the following notation. Figure 6.3 provides an example

of the following. Let F be a formula and S be a CDCL solver. We denote the full set of learned clauses derived during solving as S_L. For every conflicting state, let C' denote the clause that will be learned through conflict analysis. We let $R_{C'}$ be the set of clauses on the conflict side of the implication graph used to derive C' where $R^*_{C'} = R_{C'} \cup \bigcup_{C \in R_{C'}} R^*_C$ recursively defines the set of clauses needed to derive C' (where $R^*_{original\_clause} = \emptyset$). For every learned clause we define $D^*_{C'} = vars(C') \cup \bigcup_{C \in R^*_{C'}} D^*_C$, where $D^*_{original\_clause} = \emptyset$, as the set of variables in the clause itself as well as any learned clause used in the derivation of the clause (recursively). Intuitively, $D^*_{C'}$ is a sufficient set of *dependency variables*, such that a fresh SAT solver can absorb C' by only branching on variables in the set. For a set of clauses Δ, we let $R^*_\Delta = \bigcup_{C \in \Delta} R^*_C$ and $D^*_\Delta = \bigcup_{C \in \Delta} D^*_C$.

Lemma 6. Let S be a CDCL solver used to determine the satisfiability of some formula F. Let $\Delta \subseteq S_L$ be a set of clauses learned while solving F. Then a fresh solver S' can absorb all clauses in Δ by only branching on the variables in D^*_Δ.

Proof. We show that S' can absorb all clauses in R^*_Δ, which includes Δ. Let $seq(R^*_\Delta) = \langle C_1, C_2, \ldots C_n \rangle$ be the sequence over R^*_Δ in the order that the original solver S derived the clauses. Further, let $absorb(F, \sigma) = \Theta$ be a function that takes a formula F and sequence of clauses σ, and sequentially absorbs all clauses in σ by only branching upon variables found in clauses of σ, thus producing the learned clauses Θ. Note that this function does not exist for all inputs (F, σ). However, if for every σ_i, the clause is either 1) already absorbed; or 2) 1-empowering and 1-provable (*w.r.t.* F and all clauses learned when absorbing $\langle \sigma_1, \ldots, \sigma_{i-1} \rangle$), then we can invoke Lemma 5 to establish the existence of the function.

Consider the first clause C_1. By construction, it does not depend upon any learned clauses (i.e. it was derived from only original clauses), and since S learned C_1, it must be 1-empowering and 1-provable with respect to the initial clause set. By Lemma 5, we can absorb C_1 by only branching on variables in C_1, which again by construction are in D^*_Δ. We therefore have that $absorb(F, \langle C_1 \rangle) = \Theta_1$ absorbs C_1.

Suppose $absorb(F, \langle C_1, C_2, \ldots C_{k-1} \rangle) = \Theta_{k-1}$ absorbs the first $k-1$ clauses by only branching on the variables in $C_1, C_2, \ldots C_{k-1}$, and we wish to absorb C_k. There are two cases to consider. First, C_k may already be absorbed by Θ_{k-1}, since the clauses learned by $absorb(\ldots)$ may absorb clauses in addition to $C_1, \ldots, C_{k-1}$, in which case we are done. So suppose C_k is not absorbed by Θ_{k-1}. Since every previous clause in $seq(R^*_\Delta)$ has been absorbed, we in particular have that the clauses in R_{C_k} have been absorbed, so C_k must be 1-provable. To see this, suppose instead of absorbing R_{C_k} we learned the exact set of clauses in R_{C_k}. Then by construction, negating all literals in C_k must lead to a conflict through unit propagation. Since we have instead absorbed R_{C_k}, any propagation that was used to derive the conflict must also be possible using the clauses that absorb R_{C_k} (by definition of absorption).

We also know that C_k is 1-empowering with respect ot Θ_{k-1}, since otherwise it is absorbed by definition, and we assumed this is not true. Therefore, we can invoke Lemma 5, such that $absorb(\Theta_{k-1}, C_k) = \Theta_k$, which is derived by only branching on the variables in C_k. Again, $vars(C_k) \subseteq D^*_\Delta$ by construction. □

Theorem 7 (LSR Computation, SAT case)**.** Let S be a CDCL solver, F be a satisfiable formula, and T be the final trail of the solver immediately before returning SAT, which is composed of a sequence of decision variables T_D and propagated variables T_P. For each $p \in T_P$, let the clause used to unit propagate p be l_p and the full set of such clauses be L_P. Then $B = T_D \cup D^*_{L_p}$ constitutes an LSR backdoor for F.

Proof. Using Lemma 6, we first absorb all clauses in L_P by branching on $D^*_{L_P}$. We can then restart the solver to clear the trail, and branch on the variables in T_D, using the same order and polarity as the final trail of S. If any $d \in T_D$ is already assigned due to learned clauses used to absorb L_P, unit propagation will be able derive the literals propagated by d, since we have absorbed l_d. Note that with this final branching scheme, we cannot reach a state where the wrong polarity of a variable in T_D becomes implied through propagation (i.e. with respect to the final trail polarities), since the solver is sound and this would block the model found by the original solver S. □

Theorem 8 (LSR Computation, UNSAT case)**.** Let S be a CDCL solver, F be an unsatisfiable formula, and $\Delta \subseteq S_L$ be the set of learned clauses used to derive the final conflict. Then D^*_Δ constitutes an LSR backdoor for F.

Proof. The result follows similarly to the satisfiable case. We learn all clauses relevant to the proof using Lemma 6, which then allows unit propagation to derive UNSAT. □

We make some observations about our approach. First, our approach is not dependent upon the `1UIP` clause learning scheme, and equally applies to any asserting clause learning scheme. Second, the set of variables that constitute an LSR backdoor may be disjoint from the set of decisions made by the solver. Third, the above approach depends on the ability to restart, and therefore cannot be used to compute LS backdoors. In particular, the construction of the decision sequence for Lemma 5, as described in [Pipatsrisawat and Darwiche, 2009], requires restarting after every conflict. As an additional remark of practical importance, modern CDCL solvers often perform clause minimization to shrink a newly learned clause before adding it to the clause database [Sörensson and Biere, 2009], which can have a significant impact on performance. Intuitively, this procedure reduces the clause by finding dependencies among its literals. In order to allow clause minimization in our experiments, for each clause C we include all clauses used by the minimizer in our set R_C.

6.6 Computing Minimum LSR Backdoors

While the approach described in Section 6.5 is guaranteed to produce some LSR backdoor, there are no guarantees on its size, and the output may be arbitrarily larger than a minimum LSR backdoor. This is particularly an issue if trying to prove tight bounds on a given class of formulas. Further, since LSR backdoors are much less restrictive than previous backdoor definitions (e.g. strong backdoors), it is difficult to compute LSR backdoors manually (i.e. by hand), even for small instances. For example, we must maintain which clauses have already been learned and which variables are currently in the trail, as this may force or prohibit certain decision variables from being chosen. We further must reason about when it is favorable to restart and clear the trail.

It would therefore be useful to compute minimum LSR backdoors automatically, even if it is only feasible for small instances. While the algorithm that we describe here is quite computationally expensive, we are able to use it to compute minimum LSR backdoors for small crafted instances. Our approach is useful in the following scenarios. First, given a family of SAT formulas that scale according to some parameter, so long as it is feasible to compute minimal LSR backdoors for several increments of the parameter, the output can be used as a guide to infer LSR backdoors for arbitrarily-sized instances in the family. We demonstrate this through an example below. Further, given the construction of the algorithm, it finds the smallest length sequence of decisions (and restarts) that witnesses a backdoor. Through a slight modification to the algorithm, it can also find the minimum length sequence of decisions needed to determine the satisfiability of a formula. Finally, it can be used as a baseline to discover inefficiencies with heuristic approaches, as well as a sanity check that any LSR backdoors produced by other algorithms are not smaller than the minimal.

Our approach to computing minimum LSR backdoors is described in Algorithm 3, and works in two main steps. First, it loops over all subset of variables, in order of increasing cardinality, checking if each is a backdoor using the *isLSR* procedure. This is similar to standard approaches to computing minimal weak or strong backdoors, as in [Li and Van Beek, 2011]. To check if a given set of variables V is a backdoor, we effectively perform a breadth-first search over all possible states that the solver can reach. On Line 17, given a state s, we invoke a fresh solver with a modified branching sequence that branches according to the exact sequence of decisions given by $s.decisions$. If this sequence of decisions leads to SAT or UNSAT, we return `true`. Otherwise, on Lines 21-28, we create a new state for all possible extensions of the current state, by considering all possible decisions for the next literal. This excludes any literal already on the trail. If a state is generated that contains the same trail and set of learned clauses as a previous state, it is pruned. We ensure that any state that can be reached using k decisions over the variables in V is explored before any states that require $k+1$ decisions, for any k. When we find a final state of the solver that returns SAT or UNSAT, *isLSR()* returns true, and the set of variables V constitutes a minimum

Algorithm 3 Algorithm to compute minimum LSR backdoors.

```
1:  set<Var> minLSR(Formula F)
2:      for i = 1 to F.nVars() do
3:          for all V ⊆ F.vars() such that |V| = i do
4:              if isLSR(F, V) then
5:                  return V
6:
7:  struct State {set<Clause> learned_clauses, vector<Lit> decisions, vector<Lit> trail}
8:
9:  bool isLSR(Formula F, set<Var> V)
10:     set<State> seen
11:     queue<State> workList
12:     workList.enqueue(State(∅, [ ], [ ]))          ▷ Add the empty State to the work list.
13:     while not workList.empty() do
14:         State s = workList.pop()
15:         seen.add(s)
16:         Solver solver = new Solver(F)
17:         result = solver.replayState(s)
18:         if result == SAT or result == UNSAT then
19:             return true
20:         else                                      ▷ Generate all possible successor states.
21:             State s2 = solver.getState()
22:             for Var v ∈ V\s2.trail do             ▷ Add all extensions of s2 to workList.
23:                 State s_pos = s2.addLitToTrail(v)
24:                 State s_neg = s2.addLitToTrail(¬v)
25:                 if s_pos ∉ seen then
26:                     workList.enqueue(s_pos)
27:                 if s_neg ∉ seen then
28:                     workList.enqueue(s_neg)
29:     return false
```

Instance	LSR Backdoor	Decision Sequence
GT_4	$x_{0,1}, x_{0,2}$	$\neg x_{0,1}, x_{0,2}, \neg x_{0,2}$
GT_5	$x_{0,1}, x_{0,2}, x_{0,3}$	$\neg x_{0,1}, x_{0,2}, x_{0,3}, x_{0,3}, \neg x_{0,2}, \neg x_{0,3}$
GT_6	$x_{0,1}, x_{0,2}, x_{0,3}, x_{0,4}$	$\neg x_{0,1}, x_{0,2}, x_{0,3}, x_{0,4}, x_{0,4}, x_{0,3}, x_{0,4}, \neg x_{0,2}, \neg x_{0,3}, \neg x_{0,4}$

Table 6.1: Minimum LSR backdoors and decision sequences that witness the backdoor for several GT_n instances.

LSR backdoor.

6.6.1 Minimum LSR Backdoors for GT_n Instances

We demonstrate how we use our tool to find minimum LSR backdoors for the family of GT_n instances: unsatisfiable formulas that encode the ordering principle that any partial ordering on the set of elements $N = \{0, 1, \ldots, n-1\}$ must have a maximal element. For every $i, j \in N, i \neq j$, we introduce the variable $x_{i,j}$ which encodes the ordering predicate $i \succ j$. The formula is encoded with three sets of clauses:

$$\begin{aligned} \textit{Antisymmetry} &: \bigwedge_{i \neq j} (\neg x_{i,j} \vee \neg x_{j,i}) \\ \textit{Transitivity} &: \bigwedge_{i \neq j \neq k} (\neg x_{i,j} \vee \neg x_{j,k} \vee x_{i,k}) \\ \textit{Successor} &: \bigwedge_{i} \bigvee_{j \neq i} x_{j,i} \end{aligned} \tag{6.9}$$

The first two lines of constraints ensure that $\succ$ is a partial order, and the *Successor* constraints state that every element must have some other element greater than it, which leads to a contradiction.

Table 6.1 depicts the minimum LSR backdoor and witnessing decision sequence for GT_n for $4 \leq n \leq 6$. For $n > 6$ the approach does not terminate after 1 hour. However, given the backdoor sets for lower n, we hypothesized that a minimal LSR backdoor for arbitrary GT_n instances is $\{x_{0,i} | 1 \leq i \leq n-2\}$. Given this hypothesis, we re-ran the tool with explicit variable sets (effectively eliminating the loop on Lines 2-5), and were able to find LSR backdoors for GT_7 and GT_8, but not for higher n.

Going further, it was straightforward to manually infer a witnessing branching sequence for arbitrary n, which can be generated by Algorithm 4. We verified the sequence's correctness by overriding the SAT solver's branching heuristic to take the explicit sequence of decisions, and checked that the sequence derived unsatisfiable for all GT_n, $n \leq 100$. While this manual approach

Algorithm 4 Computes the branching sequence to witness the minimum LSR backdoor for GT_n.

```
1: list<Lit> computeGTnSequence(int n)
2:     list<Lit> L = [¬x_{0,1}]
3:     for i = 2 to n − 2 do
4:         L.append(x_{0,i})
5:     for i = n − 2 to 3 do
6:         for j = i to n − 2 do
7:             L.append(x_{0,j})
8:     for i = 2 to n − 2 do
9:         L.append(¬x_{0,i})
       return L
```

only upper-bounds the minimum LSR backdoor, we believe this to be minimum, that is, the smallest LSR backdoor for GT_n has size $n-2$.

6.7 Empirically Relating LSR Backdoors to CDCL Proofs

Finally, we show connections to LSR backdoors and the proofs generated by CDCL solvers on unsatisfiable instances. In Section 6.5, we showed that if B is the union of all variables found in the useful clauses of the proof, then B constitutes an LSR backdoor for the formula. Here, we show that frequent restarts often result in smaller and more "local" proofs, with respect to the underlying LSR backdoor. We further compare our approach to computing LSR backdoors to the algorithm used to compute upper bounds on LS backdoor sizes considered in [Dilkina et al., 2014].

We implemented our absorption-based approach to computing LSR backdoors, as described in Section 6.5. This involved modifying the off-the-shelf solver MapleSAT [Liang et al., 2016b] to annotate each learned clause C' with $D^*_{C'}$. Let $\Delta_\perp$ be the set of clauses involved the final conflict, i.e., when the solver is about to derive UNSAT. Our invariant ensures that $\bigcup_{C\in\Delta_\perp} D^*_C$ constitutes a [not-necessarily minimal] LSR backdoor. Note that we do not need to explicitly record the set $R^*_{C'}$ at any time. Different LSR backdoors can be obtained by randomizing the branching heuristic and polarity selection. However, given the size and number of instances considered here, we only perform one run per instance.

To ensure that our output is indeed an LSR backdoor, we implemented a verifier that works in three phases. First, we compute an LSR backdoor B as above. Second, we re-run the solver, and record every learned clause C such that $D^*_C \subseteq B$. We then run a final solver with a modified

branching heuristic, that iterates through the sequence of learned clauses from phase 2, absorbing each as described in Lemma 6 (first checking that the clause is either absorbed or 1-provable upon being reached in the sequence). We ensure that the solver is in a final state by the end of the sequence.

We compare several solving and restart heuristics through the lens of this spanning variables metric, which we will refer to simply as the LSR backdoor of the proof (LSR in Tables 6.2 and 6.3). Unlike the results in Chapter 4, our current experiments are conducted over only unsatisfiable instances from both the Application track of the SAT competition from 2009-2014, as well as the Agile 2016 instances.

We consider three restart policies: 1) the Luby heuristic; 2) restarting after every conflict ("Always"); and 3) never restarting. For the Agile instances, we considered three branching heuristics: LRB [Liang et al., 2016b], VSIDS [Moskewicz et al., 2001], and random branching (with phase-saving polarity selection), thus totaling 9 solver configurations in combination with the restart policies. For the Application instances, we did not include VSIDS or random branching in our experiments due to the cost of computation and to avoid the random branching heuristic greatly limiting our set of usable instances. We allotted 10,000 seconds for each Application instance, and 300 seconds for each Agile instance. Experiments were run on an Azure cluster, where each node contained two 3.1 GHz processors and 14 GB of RAM. Each experiment was limited to 6 GB. We only include instances where we could compute data for all heuristics being considered, in total, 1168 Agile instances, and 81 Application instances. For each instance, the size of the LSR backdoor is normalized by the total number of variables.

Tables 6.2 and 6.3 depict the results. On average, the always-restart policy seems to produce significantly more local proofs than the other policies, regardless of the branching heuristic. This may provide further explanation as to why restarts are useful in practice, particularly on unsatisfiable instances.

Interestingly, the always-restart policy ends up requiring the most time and conflicts to solve the Application instances; this may indicate that the usefulness of this locality is dependent on the types of instances. We also wish to emphasize that although the average LSR ratio is only 0.03 smaller for always-restart than the other policies on Application instances, this amounts to approximately 390 variables on average.

Finally, we compare our above approach to computing LSR backdoors to the previously proposed "All Decisions" approach to computing LS backdoors. In [Dilkina et al., 2009b], the authors compute LS backdoors by running a randomized non-restarting CDCL solver to completion and recording the set of all variables branched upon during search. This set constitutes an LS backdoor. The process is repeated many times to try to find small backdoors. Due to the number of instances we consider, we only use one run of the solver for each heuristic being considered.

Heuristic		Agile			
Branching	**Restart**	**LSR**	**All Decisions**	**Time (s)**	**Conflicts**
LRB	Luby	0.21 (0.08)	0.38 (0.10)	0.45 (1.81)	13392 (48874)
	Always	0.15 (0.05)	0.49 (0.13)	0.31 (1.21)	9320 (31384)
	Never	0.34 (0.15)	0.39 (0.11)	1.29 (4.25)	30450 (91745)
VSIDS	Luby	0.23 (0.09)	0.40 (0.11)	0.18 (0.73)	7783 (25684)
	Always	0.14 (0.05)	0.45 (0.12)	0.16 (0.54)	6836 (20516)
	Never	0.32 (0.14)	0.37 (0.10)	1.10 (5.23)	32665 (127482)
Random	Luby	0.76 (0.25)	0.94 (0.11)	3.12 (9.17)	52963 (138268)
	Always	0.29 (0.11)	0.96 (0.09)	2.96 (9.08)	51113 (139041)
	Never	0.75 (0.24)	0.93 (0.12)	7.63 (13.86)	107275 (174531)

Table 6.2: Depicts the average computed backdoor size (as a ratio over total variables) for each heuristic over each benchmark, with standard deviation values in parentheses. The always-restart strategy tends to produce proofs where the learned clauses span fewer variables than other strategies. This LSR backdoor approach, as described in Theorems 7 and 8, also produces smaller backdoors than the all-decisions approach. Values are normalized by the number of variables in each instance. Standard deviations are given in parentheses.

Heuristic		Application			
Branching	**Restart**	**LSR**	**All Decisions**	**Time (s)**	**Conflicts**
LRB	Luby	0.62 (0.35)	0.61 (0.35)	526.64 (931.93)	1728644 (3476414)
	Always	0.59 (0.35)	0.68 (0.33)	837.13 (1547.10)	2347710 (3935311)
	Never	0.62 (0.35)	0.60 (0.34)	682.98 (1148.91)	2544999 (5911701)

Table 6.3: Results for Application instances.

Tables 6.2 and 6.3 compare our above LSR approach to the set of all decision variables (computed on the same solver run with restarts). Since many clauses learned during search are not useful for the proof, the all-decisions approach records many unnecessary decisions that are ultimately not useful. The result does not hold on many types of crafted instances however, particularly when the formula is designed to intrinsically require proofs spanning many variables. Nonetheless, our approach seems to work for certain classes of instances found in industrial settings.

6.8 Related Work

In addition to the backdoors related work described in Section 4.6, our work is inspired by several lines of work aimed at relating the power of CDCL to general resolution. Pool resolution was first introduced by [Van Gelder, 2005] to model CDCL without restarts, and it was shown that pool resolution is exponentially stronger than regular resolution. Resolution trees with lemmas were similarly introduced in [Buss et al., 2008], and more closely match clause-learning algorithms in practice. In their seminal paper, Beame et al. formally defined CDCL as a proof system and showed that CDCL can polynomially simulate natural refinements of general resolution [Beame et al., 2011]. However, their approach required assumptions that do not reflect typical CDCL implementations, such as choosing to ignore unit propagations when preferable. In [Hertel et al., 2008], it was also shown that CDCL without restarts can effectively polynomially simulate general resolution, but required certain modifications to input formulas. In [Beame and Sabharwal, 2014], the authors showed that a non-restarting CDCL solver can polynomially simulate a restarting solver, but the approach requires adding additional variables to the formula as a "counter," based on the number of restarts performed by the original solver.

Recent approaches that show CDCL solving efficiently simulates general resolution require restarts. In their paper [Pipatsrisawat and Darwiche, 2009, Pipatsrisawat and Darwiche, 2011], the authors showed that CDCL without the assumptions from [Beame et al., 2011] can polynomially simulate general resolution. The approach relies upon the notion of *1-empowerment* [Pipatsrisawat and Darwiche, 2008], which is the dual of *clause absorption* [Atserias et al., 2011]. However, crucially, they assume that the branching and restarts in CDCL solvers are perfect (i.e., non-deterministic). In Atserias et al. [Atserias et al., 2011], the authors assume randomized branching and restarts, instead of non-deterministic ones. Specifically, they demonstrate that rapidly restarting solver with sufficiently many random decisions can effectively simulate bounded-width resolution. Many questions in this context remain open. For example, can the above simulations be modified to not require restarts? Further, can we construct realistic models of CDCL solvers (with "realistic" branching and restarts) and determine their relative power vis-a-vis well-known proof systems such as general resolution.

On the empirical side, several works have studied the performance of various restart policies. In their paper [Huang, 2007], the authors report on a comprehensive evaluation of several restart policies, which demonstrated the strength of "dynamic" restart policies such as those based on the Luby sequence [Luby et al., 1993]. [Biere and Fröhlich, 2015b] performed an evaluation of restart strategies on more modern solvers. Among their results, they showed that static restart policies can perform as well as dynamic strategies. [Haim and Heule, 2014] showed that more rapid restart policies tend to require fewer conflicts before determining satisfiability, however this does not always lead to faster solving.

6.9 Conclusions

In this chapter, we explored how the measure of backdoors relate to various solving heuristics by introducing the notion of LSR backdoors. We demonstrated an exponential separation from LS backdoors which do not allow restarts. A takeaway of this result is that clause learning together with restarts is capable of exploring the search space in ways not possible with clause learning alone. We further showed that LSR-`1UIP` backdoors may be exponentially smaller than LSR-`DL` backdoors. The order in which the search space is explored is crucial when branching over both LS and LSR backdoors, and we demonstrated several issues that may arise during the search. Empirically, we demonstrated that rapid restart strategies tend to produce significantly more local proofs than other strategies on industrial instances.

Chapter 7

Conclusions

Here we highlight the main takeaways of our work followed by some directions for future work.

7.1 Overview of Results

Before discussing some takeaways and impact from our work, we briefly outline the main technical contributions of this dissertation:

- In Chapter 3, we developed a SAT+CAS system, and used it to verify two open graph-theretic conjectures over hypercubes up to particular dimensions. We demonstrated that symmetry breaking techniques can greatly improve performance, and our approach works well compared to another SAT-based system.
- In Chapter 4, we performed a comprehensive empirical analysis of many considered measures of SAT formulas. We showed that no single parameter, nor even combinations of these features, captures the full behavior of the SAT solver, as evidenced by correlation results. However, when exploring individual sub-categories of application instances, certain measures do tend to highly correlate with solving time. Further, these measures can be used as a lens to analyze the behavior of SAT solvers and how they explore the search space. We showed that the locality of the solver, with respect to these measures, is often highly dependent upon the underlying branching heuristic and restart policy.
- In Chapter 5, we further considered the measure of mergeability. We described algorithm which takes a formula as input, and generates similar formulas with increased mergeability.

For uniformly-randomly genearated kSAT instances, we derived the expected number of merges an instance will have. We then empirically evaluated the importance of mergeability by considering a set of industrial-like randomly-generated instances, and creating series of increasingly mergeable instances. We show that mergeability strongly negatively correlates with CDCL solving time over sets of unsatisfiable instances.

- In Chapter 6, we introduced LSR backdoors, and demonstrated how this measure can be used to distinguish various branching heurstics and restart policies, both in theory and practice. We demonstrated exponential separations between solvers that allow/disallow restarts (if backjumping is disabled), and also between solvers that use the `1UIP` branching heuristic versus the decision learning heuristic. We described algorithms to compute an over-approximation of an LSR backdoor with a single run of a SAT solver, as well as more expensive algorithms to compute minimum LSR backdoors. We empirically showed that the locality of the proofs generated by the solver, in terms of the LSR backdoor size, is dependent on the restart policy, and that rapid restarts tend to produce more local proofs.

7.2 Impact and Takeaways

Our approach of combining SAT solvers with CAS has been shown to be particularly useful for combinatorial problems which can benefit from the robust search of CDCL SAT, but nonetheless require reasoning about higher-level mathematical properties easily tacked by the CAS. In addition to our graph-theoretic case studies, Bright et al. have since extended our approach primarily to explore Williamson matrices [Bright et al., 2016b, Bright et al., 2016a]. A further characterization of the benefits of SAT+CAS is detailed in [Bright, 2017] (cf. Chapter 6.1). The more general body of work of combining satisfiability checking and symbolic computation has been an emerging topic, and the SC^2 initiative, introduced by Ábrahám et al. [Ábrahám et al., 2016] around the time that MATHCHECK was released, was created to promote collaborative efforts between the two research communities. MATHCHECK-related work has since particiapted in SC^2 workshops.

Regarding the Ruskey-Savage conjecture on matchings of hypercubes (Conjecture 1), our result for Q_5 has since been mathematically verified by [Wang and Zhao, 2018], and search for the general result continues [Fink et al., 2017].

A second takeaway of our work is that these structural measures can be useful not only for traditional analyses such as runtime correlation or instance classification, but also for analyzing the behavioral properties of CDCL solvers and the underlying heuristics. In particular, these measures can describe some notion of locality in solver behavior. If such locality appears favorable in terms of solver performance, it seems reasonable to optimize the solver's heuristics to increase such

locality. Recent work on CDCL branching heuristics formulates the branching heuristic as an optimization problem, with the goal of maximizing the *global learning rate* [Liang et al., 2016b]. Their approach uses online machine learning approaches to increase this measure, however one may consider targetting other measures to be maximized, not only in the branching heuristic, but also other component of the solver such as clause learning. Ongoing collaborative work is investigating modifications to the clause learning algorithm, with the goal of learning conflict clauses that maximize the number of clause merges that occur during search.

When considering the linear correlation results between structural features and solving time, heterogenous sets of features tend to produce somewhat better correlations. Although the resulting R^2 values are still somewhat low, we believe that a good set of features must characterize both *syntactic* properties of the formula (e.g. the community structure, treewidth, and CVR), and *semantic* properties (e.g. mergeability). These types of features seem to complement each other when describing the hardness of a Boolean formula. In order to complement the mostly syntactic measures considered, we introduced mergeability, which appears to be an important feature in most of the regression models.

As a final main takeaway, our work on LSR backdoors relates to the more general problem of theoretically separating CDCL with and without restarts. This is of particular importance, since many results regarding the power of CDCL intrinsically rely upon restarts [Pipatsrisawat and Darwiche, 2009, Pipatsrisawat and Darwiche, 2011, Atserias et al., 2011]. We believe our separation is one of the first to show any theoretical separation between solvers with and without restarts. The big open question however still remains: is CDCL with restarts a more powerful proof system than CDCL without restarts? We hope that our work may be a useful step toward tackling this problem.

7.3 Future Work

Finally, we outline several questions for future work:

- *Can locality of learned clauses improve CEGAR-like algorithms, such as* MATHCHECK*?* Typically, these approaches query a SAT solver for an arbitrary model, and the refinement algorithm (e.g. the CAS in our case) confirms or refutes the model. However, if the SAT solver's heuristics are tailored to find models that are in some way similar to previously considered models, this may cut down the number of necessary iterations of the CEGAR loop.
- *Do other models beyond linear regression correlate better with CDCL solver runtime?* Our choice of linear regression was chosen for simplicity, ease-of-interpretation, and partly

based on precedent (i.e. [Newsham et al., 2014]). However, more advanced models, such as those generated by random-forest algorithms, may produce better correlations.

- *Are hybrid combinations of current features better predictors of CDCL performance?* As an example, one may combine community structure and mergeability by only considering intra-/inter-community merges.
- *Can LS and LSR backdoors be separated while allowing backjumping?* Our separation requires backtracking instead of backjumping. This direction remains unclear, particularly since a non-restarting solver effectively gets a "free restart" any time a unit clause is learned.
- *Can LS and LSR backdoors be separated for arbitrary asserting clause learning schemes?* Our current work only focuses on `1UIP`, however the result may generalize.

References

[Abio et al., 2013] Abio, I., Nieuwenhuis, R., Oliveras, A., Rodriguez-Carbonell, E., and Stuckey, P. J. (2013). To encode or to propagate? the best choice for each constraint in sat. In Schulte, C., editor, *Principles and Practice of Constraint Programming*, volume 8124 of *Lecture Notes in Computer Science*, pages 97–106. Springer Berlin Heidelberg.

[Ábrahám et al., 2016] Ábrahám, E., Abbott, J., Becker, B., Bigatti, A. M., Brain, M., Buchberger, B., Cimatti, A., Davenport, J. H., England, M., Fontaine, P., et al. (2016). Sc2: Satisfiability checking meets symbolic computation. *Intelligent Computer Mathematics: Proceedings CICM*, pages 28–43.

[Ábrahám et al., 2010] Ábrahám, E., Loup, U., Corzilius, F., and Sturm, T. (2010). A lazy SMT-solver for a non-linear subset of real algebra. *Proc. of SMT*.

[Achlioptas et al., 2001] Achlioptas, D., Beame, P., and Molloy, M. (2001). A sharp threshold in proof complexity. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 337–346. ACM.

[Aloul et al., 2003] Aloul, F. A., Markov, I. L., and Sakallah, K. A. (2003). SHATTER: efficient symmetry-breaking for boolean satisfiability. In *Proceedings of the 40th annual Design Automation Conference*, pages 836–839. ACM.

[Ansótegui et al., 2014] Ansótegui, C., Bonet, M. L., Giráldez-Cru, J., and Levy, J. (2014). The fractal dimension of sat formulas. In *International Joint Conference on Automated Reasoning*, pages 107–121. Springer.

[Ansótegui et al., 2017] Ansótegui, C., Bonet, M. L., Giráldez-Cru, J., and Levy, J. (2017). Structure features for sat instances classification. *Journal of Applied Logic*, 23:27–39.

[Ansótegui et al., 2009] Ansótegui, C., Bonet, M. L., and Levy, J. (2009). On the structure of industrial sat instances. In *International Conference on Principles and Practice of Constraint Programming*, pages 127–141. Springer.

[Ansótegui et al., 2012] Ansótegui, C., Giráldez-Cru, J., and Levy, J. (2012). The community structure of SAT formulas. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 410–423. Springer.

[Ansótegui et al., 2015] Ansótegui, C., Giráldez-Cru, J., Levy, J., and Simon, L. (2015). Using community structure to detect relevant learned clauses. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 238–254. Springer.

[Ansotegui et al., 2015] Ansotegui, C., Giraldez-Cru, J., Levy, J., and Simon, L. (2015). Using Community Structure to Detect Relevant Learnt Clauses. *Theory and Applications of Satisfiability Testing - Sat 2015*, 9340:238–254.

[Areces et al., 2013] Areces, C., Déharbe, D., Fontaine, P., and Orbe, E. (2013). SYMT: finding symmetries in SMT formulas. In *SMT Workshop 2013 11th International Workshop on Satisfiability Modulo Theories*.

[Armand et al., 2011] Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., and Wener, B. (2011). Verifying sat and smt in coq for a fully automated decision procedure. In *PSATTT'11: International Workshop on Proof-Search in Axiomatic Theories and Type Theories*.

[Atserias et al., 2011] Atserias, A., Fichte, J. K., and Thurley, M. (2011). Clause-learning algorithms with many restarts and bounded-width resolution. *Journal of Artificial Intelligence Research*, 40:353–373.

[Audemard and Simon, 2009] Audemard, G. and Simon, L. (2009). Predicting learnt clauses quality in modern sat solvers. In *IJCAI*, volume 9, pages 399–404.

[Barrett et al., 2011] Barrett, C., Conway, C., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., and Tinelli, C. (2011). Cvc4. In Gopalakrishnan, G. and Qadeer, S., editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer Berlin Heidelberg.

[Batory, 2005] Batory, D. (2005). Feature models, grammars, and propositional formulas. In *International Conference on Software Product Lines*, pages 7–20. Springer.

[Bayless et al., 2015] Bayless, S., Bayless, N., Hoos, H. H., and Hu, A. J. (2015). SAT modulo monotonic theories. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*.

[Beame et al., 2011] Beame, P., Kautz, H., and Sabharwal, a. (2011). Towards Understanding and Harnessing the Potential of Clause Learning. *Artificial Intelligence*, 22:319–351.

[Beame and Sabharwal, 2014] Beame, P. and Sabharwal, A. (2014). Non-restarting sat solvers with simple preprocessing can efficiently simulate resolution. In *AAAI Conference on Artificial Intelligence*, pages 2608–2615. AAAI Press.

[Belov et al., 2014] Belov, A., Diepol, D., Heule, M., and Järvisalo, M. (2014). Sat competition 2014.

[Benhamou et al., 2010] Benhamou, B., Nabhani, T., Ostrowski, R., and Saïdi, M. R. (2010). Enhancing clause learning by symmetry in SAT solvers. In *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on*, volume 1, pages 329–335. IEEE.

[Biere, 2016] Biere, A. (2016). Collection of Combinational Arithmetic Miters Submitted to the SAT Competition 2016. In Balyo, T., Heule, M., and Järvisalo, M., editors, *Proc. of SAT Competition 2016 – Solver and Benchmark Descriptions*, volume B-2016-1 of *Department of Computer Science Series of Publications B*, pages 65–66. University of Helsinki.

[Biere and Fröhlich, 2015a] Biere, A. and Fröhlich, A. (2015a). Evaluating CDCL restart schemes. In *Pragmatics of SAT workshop*.

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

[Biere et al., 2009a] Biere, A., Heule, M., and van Maaren, H. (2009a). *Handbook of satisfiability*, volume 185. IOS press.

[Biere et al., 2009b] Biere, A., Heule, M. J. H., van Maaren, H., and Walsh, T., editors (2009b). *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

[Blondel et al., 2008] Blondel, V. D., Guillaume, J.-L., Lambiotte, R., and Lefebvre, E. (2008). Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008.

[Bonet and Buss, 2013] Bonet, M. L. and Buss, S. (2013). An improved separation of regular resolution from pool resolution and clause learning. *IJCAI International Joint Conference on Artificial Intelligence*, pages 2972–2976.

[Bosma et al., 1997] Bosma, W., Cannon, J., and Playoust, C. (1997). The MAGMA algebra system I: The user language. *Journal of Symbolic Computation*, 24(3):235–265.

[Bouton et al., 2009] Bouton, T., De Oliveira, D. C. B., Déharbe, D., and Fontaine, P. (2009). VERIT: an open, trustable and efficient SMT-solver. In Schmidt, R. A., editor, *Automated Deduction – CADE-22*, volume 5663 of *LNCS*, pages 151–156. Springer Berlin Heidelberg.

[Bright, 2017] Bright, C. (2017). Computational methods for combinatorial and number theoretic problems.

[Bright et al., 2016a] Bright, C., Ganesh, V., Heinle, A., Kotsireas, I., Nejati, S., and Czarnecki, K. (2016a). Mathcheck2: A sat+ cas verifier for combinatorial conjectures. In *International Workshop on Computer Algebra in Scientific Computing*, pages 117–133. Springer.

[Bright et al., 2016b] Bright, C., Ganesh, V., Heinle, A., lias Kotsireas, Nejati, S., and Czarnecki, K. (2016b). MATHCHECK2: A SAT+CAS verifier for combinatorial conjectures. In *Computer Algebra in Scientific Computing (to appear)*. Springer Berlin Heidelberg.

[Buss et al., 2008] Buss, S. R., Hoffmann, J., and Johannsen, J. (2008). Resolution trees with lemmas: Resolution refinements that characterize dll algorithms with clause learning. *arXiv preprint arXiv:0811.1075*.

[Char et al., 1986] Char, B. W., Fee, G. J., Geddes, K. O., Gonnet, G. H., and Monagan, M. B. (1986). A tutorial introduction to MAPLE. *Journal of Symbolic Computation*, 2(2):179–200.

[Chen and Li, 2010] Chen, Y.-C. and Li, K.-L. (2010). Matchings extend to perfect matchings on hypercube networks. In *Proceedings of the International MultiConference of Engineers and Computer Scientists*, volume 1. Citeseer.

[Clarke et al., 2000] Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. (2000). Counterexample-guided abstraction refinement. In *Computer aided verification*, pages 154–169. Springer.

[Coarfa et al., 2000] Coarfa, C., Demopoulos, D. D., Aguirre, A. S. M., Subramanian, D., and Vardi, M. Y. (2000). Random 3-sat: The plot thickens. In *International Conference on Principles and Practice of Constraint Programming*, pages 143–159. Springer.

[Cok et al., 2014] Cok, D. R., Stump, A., and Weber, T. (2014). The 2013 SMT Evaluation.

[Colbourn and Dinitz, 2007] Colbourn, C. J. and Dinitz, J. H., editors (2007). *Handbook of Combinatorial Designs*. Discrete Mathematics and its Applications (Boca Raton). Chapman & Hall/CRC, Boca Raton, FL, second edition.

[Courcelle et al., 2001] Courcelle, B., Makowsky, J. A., and Rotics, U. (2001). On the fixed parameter complexity of graph enumeration problems definable in monadic second-order logic. *Discrete Applied Mathematics*, 108(1):23–52.

[Darga et al., 2008] Darga, P. T., Sakallah, K. A., and Markov, I. L. (2008). Faster symmetry discovery using sparsity of symmetries. In *Proceedings of the 45th annual Design Automation Conference*, pages 149–154. ACM.

[De Moura and Bjørner, 2008] De Moura, L. and Bjørner, N. (2008). Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer.

[Déharbe et al., 2011] Déharbe, D., Fontaine, P., Merz, S., and Paleo, B. W. (2011). Exploiting symmetry in SMT problems. In *Automated Deduction–CADE-23*, pages 222–236. Springer.

[Dilkina et al., 2009a] Dilkina, B., Gomes, C., Malitsky, Y., Sabharwal, A., and Sellmann, M. (2009a). Backdoors to Combinatorial Optimization: Feasibility and Optimality. *CPAIOR*, 5547:56 – 70.

[Dilkina et al., 2009b] Dilkina, B., Gomes, C., and Sabharwal, A. (2009b). Backdoors in the Context of Learning. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 73 – 79. Springer.

[Dilkina et al., 2014] Dilkina, B., Gomes, C. P., and Sabharwal, A. (2014). Tradeoffs in the complexity of backdoors to satisfiability: dynamic sub-solvers and learning during search. *Annals of Mathematics and Artificial Intelligence*, 70(4):399–431.

[Dooms et al., 2005] Dooms, G., Deville, Y., and Dupont, P. (2005). Cp (graph): Introducing a graph computation domain in constraint programming. In *Principles and Practice of Constraint Programming-CP 2005*, pages 211–225. Springer.

[Downey and Fellows, 2013] Downey, R. G. and Fellows, M. R. (2013). *Fundamentals of parameterized complexity*, volume 4. Springer.

[Durán et al., 2014] Durán, A. J., Pérez, M., and Varona, J. L. (2014). The misfortunes of a trio of mathematicians using computer algebra systems. can we trust in them? *Notices of the AMS*, 61(10).

[Eén and Sörensson, 2003] Eén, N. and Sörensson, N. (2003). An extensible SAT-solver. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 502–518. Springer.

[Een and Sörensson, 2005] Een, N. and Sörensson, N. (2005). MINISAT: A SAT solver with conflict-clause minimization. *Sat*, 5:8th.

[Feder and Subi, 2013] Feder, T. and Subi, C. (2013). On hypercube labellings and antipodal monochromatic paths. *Discrete Applied Mathematics*, 161(10):1421–1426.

[Fichte and Szeider, 2015] Fichte, J. K. and Szeider, S. (2015). Backdoors to tractable answer set programming. *Artificial Intelligence*, 220(0):64–103.

[Fink, 2007] Fink, J. (2007). Perfect matchings extend to hamilton cycles in hypercubes. *Journal of Combinatorial Theory, Series B*, 97(6):1074–1076.

[Fink, 2009] Fink, J. (2009). Connectivity of matching graph of hypercube. *SIAM Journal on Discrete Mathematics*, 23(2):1100–1109.

[Fink et al., 2017] Fink, J., Dvořák, T., Gregor, P., and Novotnỳ, T. (2017). Towards a problem of ruskey and savage on matching extendability. *Electronic Notes in Discrete Mathematics*, 61:437–443.

[Fletcher et al., 2001] Fletcher, R. J., Gysin, M., and Seberry, J. (2001). Application of the discrete Fourier transform to the search for generalised Legendre pairs and Hadamard matrices. *Faculty of Informatics-Papers*, page 317.

[Fontaine et al., 2010] Fontaine, P., Merz, S., and Paleo, B. W. (2010). Exploring and exploiting algebraic and graphical properties of resolution. In *8th International Workshop on Satisfiability Modulo Theories-SMT 2010*.

[Frigo and Johnson, 2005] Frigo, M. and Johnson, S. G. (2005). The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[Frohlich et al., 2015] Frohlich, A., Biere, A., Wintersteiger, C. M., and Hamadi, Y. (2015). Stochastic Local Search for Satisfiability Modulo Theories. *Aaai-2015*.

[Ganesh and Dill, 2007] Ganesh, V. and Dill, D. L. (2007). A decision procedure for bit-vectors and arrays. In *Computer Aided Verification*, pages 519–531. Springer.

[Ganesh et al., 2012] Ganesh, V., O'donnell, C. W., Soos, M., Devadas, S., Rinard, M. C., and Solar-Lezama, A. (2012). Lynx: A programmatic sat solver for the rna-folding problem. In *Theory and Applications of Satisfiability Testing–SAT 2012*, pages 143–156. Springer.

[Ganian et al., 2017] Ganian, R., Ramanujan, M., and Szeider, S. (2017). Backdoor treewidth for sat. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 20–37. Springer.

[Ganian and Szeider, 2015a] Ganian, R. and Szeider, S. (2015a). Community Structure Inspired Algorithms for SAT and # SAT. *Sat 2015 (Lncs9340)*, (September):223–237.

[Ganian and Szeider, 2015b] Ganian, R. and Szeider, S. (2015b). Community structure inspired algorithms for sat and# sat. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 223–237. Springer.

[Gebser et al., 2014] Gebser, M., Janhunen, T., and Rintanen, J. (2014). Sat modulo graphs: Acyclicity. In *Logics in Artificial Intelligence*, pages 137–151. Springer.

[Gent et al., 2006] Gent, I. P., Petrie, K. E., and Puget, J.-F. (2006). Symmetry in Constraint Programming. *Handbook of Constraint Programming*, pages 329–376.

[Gent and Smith, 1999] Gent, I. P. and Smith, B. (1999). *Symmetry breaking during search in constraint programming*. Citeseer.

[Gini, 1921] Gini, C. (1921). Measurement of inequality of incomes. *The Economic Journal*, 31(121):124–126.

[Giráldez-Cru and Levy, 2017] Giráldez-Cru, J. and Levy, J. (2017). Locality in random sat instances. In *Proc. of the 26th Int. Joint Conf. on Artificial Intelligence (IJCAI'17)*.

[Godefroid et al., 2008] Godefroid, P., Levin, M. Y., Molnar, D. A., et al. (2008). Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium*, pages 151–166. Internet Society.

[Gregor, 2009] Gregor, P. (2009). Perfect matchings extending on subcubes to hamiltonian cycles of hypercubes. *Discrete Mathematics*, 309(6):1711–1713.

[Gregory et al., 2008] Gregory, P., Fox, M., and Long, D. (2008). A new empirical study of weak backdoors. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5202 LNCS:618–623.

[Hadamard, 1893] Hadamard, J. (1893). Résolution d'une question relative aux déterminants. *Bull. Sci. Math.*, 17(1):240–246.

[Haim and Heule, 2014] Haim, S. and Heule, M. (2014). Towards ultra rapid restarts. *arXiv preprint arXiv:1402.4413*.

[Hall et al., 2009] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18.

[Hedayat and Wallis, 1978] Hedayat, A. and Wallis, W. (1978). Hadamard matrices and their applications. *The Annals of Statistics*, 6(6):1184–1238.

[Hertel et al., 2008] Hertel, P., Bacchus, F., Pitassi, T., and Van Gelder, A. (2008). Clause learning can effectively p-simulate general propositional resolution. In *AAAI Conference on Artificial Intelligence*, pages 283–290. AAAI Press.

[Heule et al., 2013] Heule, M. J., Hunt, W., and Wetzler, N. (2013). Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 181–188. IEEE.

[Heule et al., 2016a] Heule, M. J., Kullmann, O., and Marek, V. W. (2016a). Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. *arXiv preprint arXiv:1605.00723*.

[Heule and Biere, 2013] Heule, M. J. H. and Biere, A. (2013). Proofs for Satisfiability Problems.

[Heule et al., 2016b] Heule, M. J. H., Kullmann, O., and Marek, V. W. (2016b). *Solving and Verifying the Boolean Pythagorean Triples Problem via Cube-and-Conquer*, pages 228–245. Springer International Publishing, Cham.

[Holm et al., 2001] Holm, J., De Lichtenberg, K., and Thorup, M. (2001). Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM (JACM)*, 48(4):723–760.

[Huang, 2007] Huang, J. (2007). The effect of restarts on the efficiency of clause learning. pages 2318–2323. AAAI Press.

[Jackson, 2012] Jackson, D. (2012). *Software Abstractions: logic, language, and analysis*. MIT press.

[Janota et al., 2015] Janota, M., Lynce, I., and Marques-Silva, J. (2015). Algorithms for computing backbones of propositional formulae. *AI Communications*, 28(2):161–177.

[Järvisalo et al., 2012] Järvisalo, M., Matsliah, A., Nordström, J., and Živnỳ, S. (2012). Relating proof complexity measures and practical hardness of sat. In *Principles and Practice of Constraint Programming*, pages 316–331. Springer.

[Järvisalo and Niemelä, 2008] Järvisalo, M. and Niemelä, I. (2008). The effect of structural branching on the efficiency of clause learning SAT solving: An experimental study. *Journal of Algorithms*, 63(1-3):90–113.

[Jordi, 2015] Jordi, L. (2015). On the Classification of Industrial SAT Families. In *International Conference of the Catalan Association for Artificial Intelligence*, page 163. IOS Press.

[Junttila and Kaski, 2007] Junttila, T. and Kaski, P. (2007). Engineering an efficient canonical labeling tool for large and sparse graphs. In Applegate, D., Brodal, G. S., Panario, D., and Sedgewick, R., editors, *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics*, pages 135–149. SIAM.

[Katsirelos and Bacchus, 2005] Katsirelos, G. and Bacchus, F. (2005). Generalized nogoods in csps. In *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 390–396.

[Kilby et al., 2005] Kilby, P., Slaney, J., Thiébaux, S., and Walsh, T. (2005). Backbones and backdoors in satisfiability. In *AAAI Conference on Artificial Intelligence*, pages 1368–1373. AAAI Press.

[Kolb et al., 2010] Kolb, H., Pechenik, O., and Wise, J. (2010). Antipodal edge-colorings of hypercubes.

[Konev and Lisitsa, 2014] Konev, B. and Lisitsa, A. (2014). A SAT attack on the Erdős discrepancy conjecture. In *SAT*.

[Kotsireas, 2013] Kotsireas, I. S. (2013). Algorithms and Metaheuristics for Combinatorial Matrices. In *Handbook of Combinatorial Optimization*, pages 283–309. Springer New York.

[Kotsireas et al., 2006a] Kotsireas, I. S., Koukouvinos, C., and Seberry, J. (2006a). Hadamard ideals and Hadamard matrices with circulant core.

[Kotsireas et al., 2006b] Kotsireas, I. S., Koukouvinos, C., and Seberry, J. (2006b). Hadamard ideals and Hadamard matrices with two circulant cores. *European Journal of Combinatorics*, 27(5):658–668.

[Koukouvinos and Kounias, 1988] Koukouvinos, C. and Kounias, S. (1988). Hadamard matrices of the Williamson type of order $4 \cdot m$, $m = p \cdot q$ an exhaustive search for $m = 33$. *Discrete mathematics*, 68(1):45–57.

[Levy and Csic, 2015] Levy, J. and Csic, I. (2015). A Modularity-Based Random SAT Instances Generator. (Ijcai):1952–1958.

[Li and Van Beek, 2011] Li, Z. and Van Beek, P. (2011). Finding small backdoors in SAT instances. In *Canadian Conference on Artificial Intelligence*, pages 269–280. Springer.

[Liang et al., 2015a] Liang, J. H., Ganesh, V., Czarnecki, K., and Raman, V. (2015a). SAT-based analysis of large real-world feature models is easy. In *International Conference on Software Product Line*, pages 91–100. ACM.

[Liang et al., 2016a] Liang, J. H., Ganesh, V., Poupart, P., and Czarnecki, K. (2016a). Exponential recency weighted average branching heuristic for SAT solvers. In [Schuurmans and Wellman, 2016], pages 3434–3440.

[Liang et al., 2016b] Liang, J. H., Ganesh, V., Poupart, P., and Czarnecki, K. (2016b). Learning rate based branching heuristic for SAT solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 123–140. Springer.

[Liang et al., 2015b] Liang, J. H., Ganesh, V., Zulkoski, E., Zaman, A., and Czarnecki, K. (2015b). Understanding VSIDS branching heuristics in conflict-driven clause-learning SAT solvers. In *Haifa Verification Conference*, pages 225–241. Springer.

[Liang et al., 2016c] Liang, J. H., Oh, C., Ganesh, V., Czarnecki, K., and Poupart, P. (2016c). Maple-comsps, maplecomsps lrb, maplecomsps chb. *SAT Competition*, page 52.

[Lifschitz, 2008] Lifschitz, V. (2008). What is answer set programming?. In *AAAI*, volume 8, pages 1594–1597.

[Lonsing and Biere, 2011] Lonsing, F. and Biere, A. (2011). Failed literal detection for qbf. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 259–272. Springer.

[Luby et al., 1993] Luby, M., Sinclair, A., and Zuckerman, D. (1993). Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47(4):173–180.

[Manolios and Papavasileiou, 2011] Manolios, P. and Papavasileiou, V. (2011). Pseudo-boolean solving by incremental translation to sat. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, pages 41–45. FMCAD Inc.

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

[Marx et al., 2013] Marx, O., Wedler, M., Stoffel, D., Kunz, W., and Dreyer, A. (2013). Proof logging for computer algebra based smt solving. In *Proceedings of the International Conference on Computer-Aided Design*, pages 677–684. IEEE Press.

[Mateescu, 2011] Mateescu, R. (2011). Treewidth in Industrial SAT Benchmarks.

[Matti et al., 2012] Matti, J., Matsliah, A., Nordstr, J., and Zivn, S. (2012). Relating Proof Complexity Measures and Practical Hardness of SAT. (Cdcl):316–331.

[Milicevic et al., 2015] Milicevic, A., Near, J. P., Kang, E., and Jackson, D. (2015). ALLOY*: A general-purpose higher-order relational constraint solver. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 609–619.

[Monasson et al., 1999] Monasson, R., Zecchina, R., Kirkpatrick, S., Selman, B., and Troyansky, L. (1999). Determining computational complexity from characteristic 'phase transitions'. *Nature*, 400(6740):133–137.

[Moskewicz et al., 2001] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. (2001). Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535. ACM.

[Mull et al., 2016] Mull, N., Fremont, D. J., and Seshia, S. A. (2016). On the hardness of sat with community structure. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 141–159. Springer.

[Muller, 1954] Muller, D. E. (1954). Application of Boolean Algebra to Switching Circuit Design and to Error Detection. *Electronic Computers, Transactions of the IRE Professional Group on Electronic Computers*, EC-3(3):6–12.

[Nejati et al., 2016] Nejati, S., Liang, J. H., Ganesh, V., Gebotys, C., and Czarnecki, K. (2016). Adaptive restart and cegar-based solver for inverting cryptographic hash functions. *arXiv preprint arXiv:1608.04720*.

[Newsham et al., 2014] Newsham, Z., Ganesh, V., Fischmeister, S., Audemard, G., and Simon, L. (2014). Impact of community structure on SAT solver performance. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 252–268. Springer.

[Nieuwenhuis et al., 2004] Nieuwenhuis, R., Oliveras, A., and Tinelli, C. (2004). Abstract dpll and abstract dpll modulo theories. In Baader, F. and Voronkov, A., editors, *LPAR*, volume 3452 of *Lecture Notes in Computer Science*, pages 36–50. Springer.

[Nipkow et al., 2002] Nipkow, T., Paulson, L. C., and Wenzel, M. (2002). ISABELLE/HOL*: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media.

[Nishimura et al., 2004] Nishimura, N., Ragde, P., and Szeider, S. (2004). Detecting Backdoor Sets with Respect to {Horn} and Binary Clauses. *Proceedings of SAT 2004 (Seventh International Conference on Theory and Applications of Satisfiability Testing, 10–13 May, 2004, Vancouver, BC, Canada)*, pages 96–103.

[Norine, 2008] Norine, S. (2008). Edge-antipodal Colorings of Cubes. Open Problems Garden.

[Oh, 2015] Oh, C. (2015). Between SAT and UNSAT: the fundamental difference in CDCL SAT. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 307–323. Springer.

[Oh, 2016] Oh, C. (2016). Improving SAT Solvers by Exploiting Empirical Characteristics of CDCL. (January).

[Ohrimenko et al., 2009] Ohrimenko, O., Stuckey, P. J., and Codish, M. (2009). Propagation via lazy clause generation. *Constraints*, 14(3):357–391.

[Đoković, 1993] Đoković, D. Ž. (1993). Williamson matrices of order $4n$ for $n = 33, 35, 39$. *Discrete mathematics*, 115(1):267–271.

[Đoković and Kotsireas, 2015] Đoković, D. Ž. and Kotsireas, I. S. (2015). Compression of periodic complementary sequences and applications. *Designs, Codes and Cryptography*, 74(2):365–377.

[Paley, 1933] Paley, R. E. (1933). On Orthogonal Matrices. *J. Math. Phys.*, 12(1):311–320.

[Pipatsrisawat and Darwiche, 2008] Pipatsrisawat, K. and Darwiche, A. (2008). A new clause learning scheme for efficient unsatisfiability proofs. In *AAAI Conference on Artificial Intelligence*, pages 1481–1484.

[Pipatsrisawat and Darwiche, 2009] Pipatsrisawat, K. and Darwiche, A. (2009). On the power of clause-learning SAT solvers with restarts. In *International Conference on Principles and Practice of Constraint Programming*, pages 654–668. Springer.

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

[Ramos et al., 2011] Ramos, A., Van Der Tak, P., and Heule, M. J. (2011). Between restarts and backjumps. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 216–229. Springer.

[Reed, 1954] Reed, I. (1954). A Class of Multiple-Error-Correcting Codes and the Decoding Scheme. *Transactions of the IRE Professional Group on Information Theory*, 4(4):38–49.

[Robertson and Seymour, 1984] Robertson, N. and Seymour, P. D. (1984). Graph minors. III. Planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64.

[Ruan et al., 2004] Ruan, Y., Kautz, H., and Horvitz, E. (2004). The Backdoor Key : A Path to Understanding Problem Hardness. *Aaai 2004*.

[Ruskey and Savage, 1993] Ruskey, F. and Savage, C. (1993). Hamilton cycles that extend transposition matchings in cayley graphs of s_n. *SIAM Journal on Discrete Mathematics*, 6(1):152–166.

[Sakallah, 2009] Sakallah, K. A. (2009). Symmetry and satisfiability. *Handbook of Satisfiability*, 185:289–338.

[Samer and Szeider, 2008] Samer, M. and Szeider, S. (2008). Backdoor Trees. *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence - AAAI'2008*, pages 363–368.

[SAT, 2017] SAT (2017). *The international SAT Competitions web page*. http://www.satcompetition.org/.

[Schuurmans and Wellman, 2016] Schuurmans, D. and Wellman, M. P., editors (2016). *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*. AAAI Press.

[Sebastiani, 2007] Sebastiani, R. (2007). Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 3:141–224.

[Seberry and Yamada, 1990] Seberry, J. and Yamada, M. (1990). On the products of hadamard matrices, williamson matrices and other orthogonal matrices using m-structures.

[Selman et al., 1996] Selman, B., Mitchell, D. G., and Levesque, H. J. (1996). Generating hard satisfiability problems. *Artificial intelligence*, 81(1-2):17–29.

[SHARCNET, 2017] SHARCNET (2017). *SHARCNET*. http://www.sharcnet.ca/.

[Simon, 2014] Simon, L. (2014). Post mortem analysis of sat solver proofs. In *POS@ SAT*, pages 26–40.

[Sloane, 1999] Sloane, N. J. (1999). A library of hadamard matrices. *available at the website: http://www. research. att. com/˜ njas/hadamard*.

[Soh et al., 2014] Soh, T., Le Berre, D., Roussel, S., Banbara, M., and Tamura, N. (2014). Incremental sat-based method with native boolean cardinality handling for the hamiltonian cycle problem. In *Logics in Artificial Intelligence*, pages 684–693. Springer.

[Sörensson and Biere, 2009] Sörensson, N. and Biere, A. (2009). Minimizing learned clauses. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 237–243. Springer.

[Stein and Etal., 2010] Stein, W. A. and Etal. (2010). Sage Mathematics Software (Version 6.3).

[Sylvester, 1867] Sylvester, J. J. (1867). Thoughts on inverse orthogonal matrices, simultaneous sign successions, and tessellated pavements in two or more colours, with applications to Newton's rule, ornamental tile-work, and the theory of numbers. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 34(232):461–475.

[Szeider, 2003] Szeider, S. (2003). On fixed-parameter tractable parameterizations of SAT. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 188–202. Springer.

[Szeider, 2006] Szeider, S. (2006). Backdoor Sets for DLL Subsolvers. *Journal of Automated Reasoning*, 35(1-3):73–88.

[COQ development team, 2004] COQ development team, T. (2004). *The* COQ *proof assistant reference manual*. LogiCal Project. Version 8.0.

[Thurley, 2006] Thurley, M. (2006). sharpsat–counting models with advanced component caching and implicit bcp. In *Theory and Applications of Satisfiability Testing-SAT 2006*, pages 424–429. Springer.

[Torlak, 2009] Torlak, E. (2009). *A constraint solver for software engineering: finding models and cores of large relational specifications*. PhD thesis, Massachusetts Institute of Technology.

[Tseitin, 1968] Tseitin, G. (1968). On the complexity ofderivation in propositional calculus. *Studies in Constrained Mathematics and Mathematical Logic*.

[Van Gelder, 2005] Van Gelder, A. (2005). Pool resolution and its relation to regular resolution and DPLL with clause learning. pages 580–594. Springer.

[Velev and Gao, 2009] Velev, M. N. and Gao, P. (2009). Efficient sat techniques for absolute encoding of permutation problems: Application to hamiltonian cycles. In *SARA*.

[Walsh, 1923] Walsh, J. L. (1923). A Closed Set of Normal Orthogonal Functions. *American Journal of Mathematics*, 45(1):5–24.

[Walt et al., 2011] Walt, S. v. d., Colbert, S. C., and Varoquaux, G. (2011). The NUMPY array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30.

[Wang and Zhao, 2018] Wang, F. and Zhao, W. (2018). Matchings extend to hamiltonian cycles in 5-cube. *Discussiones Mathematicae Graph Theory*, 38(1):217–231.

[Wang, 2015] Wang, G. (2015). H Eterogeneous C Loud R Adio a Ccess N Etworks a Cquisition of C Hannel S Tate I Nformation in H Eterogeneous C Loud R Adio a Ccess N Etworks : C Hallenges and R Esearch D Irections. (June):100–107.

[Wetzler et al., 2014] Wetzler, N., Heule, M. J., and Hunt Jr, W. A. (2014). DRAT-TRIM: Efficient checking and trimming using expressive clausal proofs. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 422–429. Springer.

[Williams et al., 2003a] Williams, R., Gomes, C., and Selman, B. (2003a). On the connections between backdoors, restarts, and heavy-tailedness in combinatorial search. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 222–230. Springer.

[Williams et al., 2003b] Williams, R., Gomes, C. P., and Selman, B. (2003b). Backdoors to typical case complexity. In *International Joint Conference on Artificial Intelligence*, number 24, pages 1173–1178. AAAI Press.

[Williamson, 1944] Williamson, J. (1944). Hadamard's Determinant Theorem and the Sum of Four Squares. *Duke Math. J*, 11(1):65–81.

[Wolfram, 1999] Wolfram, S. (1999). *The* MATHEMATICA *Book, version 4*. Cambridge University Press.

[Xu et al., 2008] Xu, L., Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2008). SATzilla: portfolio-based algorithm selection for SAT. *Journal of artificial intelligence research*, 32:565–606.

[Zhang et al., 2001] Zhang, L., Madigan, C. F., Moskewicz, M. H., and Malik, S. (2001). Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285. IEEE Press.

[Zulkoski, 2017] Zulkoski, E. (2017). *Supplemental Material*. https://bitbucket.org/ezulkosk/backdoors.

[Zulkoski et al., 2017a] Zulkoski, E., Bright, C., Heinle, A., Kotsireas, I., Czarnecki, K., and Ganesh, V. (2017a). Combining sat solvers with computer algebra systems to verify combinatorial conjectures. *Journal of Automated Reasoning*, 58(3):313–339.

[Zulkoski and Ganesh, 2015] Zulkoski, E. and Ganesh, V. (2015). SageSAT. https://bitbucket.org/ezulkosk/sagesat.

[Zulkoski and Ganesh, 2017] Zulkoski, E. and Ganesh, V. (2017). Mergeability Generator. https://github.com/ezulkosk/proof_graph_analyzer.

[Zulkoski et al., 2015] Zulkoski, E., Ganesh, V., and Czarnecki, K. (2015). MATHCHECK: A math assistant based on a combination of computer algebra systems and SAT solvers. In *International Conference on Automated Deduction*, Berlin, Germany. Springer, Springer.

[Zulkoski et al., 2016] Zulkoski, E., Ganesh, V., and Czarnecki, K. (2016). Mathcheck: a math assistant via a combination of computer algebra systems and sat solvers. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, pages 4228–4232. AAAI Press.

[Zulkoski et al., 2017b] Zulkoski, E., Martins, R., and Ganesh, V. (2017b). Laser. https://github.com/sat-group/laser-maple.

[Zulkoski et al., 2018a] Zulkoski, E., Martins, R., Wintersteiger, C., Liang, J., Czarnecki, K., and Ganesh, V. (2018a). The effect of structural measures and merges on sat solver performance.

[Zulkoski et al., 2017c] Zulkoski, E., Martins, R., Wintersteiger, C., Robere, R., Liang, J., Czarnecki, K., and Ganesh, V. (2017c). Relating complexity-theoretic parameters with sat solver performance. *CoRR*, abs/1706.08611.

[Zulkoski et al., 2017d] Zulkoski, E., Martins, R., Wintersteiger, C., Robert, R., Liang, J., Czarnecki, K., and Ganesh, V. (2017d). Empirically relating complexity theoretic parameters with sat solver performance. In *Pragmatics of Constraint Reasoning*.

[Zulkoski et al., 2018b] Zulkoski, E., Martins, R., Wintersteiger, C., Robert, R., Liang, J., Czarnecki, K., and Ganesh, V. (2018b). Learning sensitive backdoors with restarts.

APPENDICES

Appendix A

Additional Correlation Results

Here, we provide more details regarding the correlation results in Chapter 4. In Table A.1, we repeat our correlation experiment using ridge regression and MapleCOMSPS as the solver, but only consider satisfiable instances. The same experiment is repeated in Table A.2 for unsatisfiable instances. In Table A.3, we use Lingeling as our solver for runtimes and linear regression (over all instances), and in Table A.4 we repeat the experiment using ridge regression. Table A.5 displays the Pearson and Spearman correlations between various measures and Lingeling solving time, split by sub-category. Tables A.6 - A.9 display the coefficient values of the best linear regression models found using MapleCOMSPS as the solver, for each benchmark.

Feature Set	Application	Crafted	Random	Agile
$V \oplus C \oplus C/V$	0.02 (422)	0.08 (236)	0.02 (76)	0.10 (315)
$V \oplus C \oplus Cmtys \oplus Q \oplus Q/Cmtys$	0.09 (311)	0.15 (227)	0.05 (76)	0.27 (315)
$V \oplus C \oplus LSR \oplus LSR/V$	0.30 (387)	0.37 (223)	0.08 (75)	0.22 (241)
$V \oplus C \oplus \#Min\_Weak \oplus Weak$	0.03 (267)	0.12 (195)	0.06 (76)	0.15 (101)
$V \oplus C \oplus Bones \oplus Bones/V$	0.20 (190)	0.31 (153)	0.08 (59)	0.14 (208)
$V \oplus C \oplus TW \oplus TW/V$	0.03 (379)	0.09 (236)	0.02 (76)	0.17 (315)
$V \oplus C \oplus Dim_V \oplus Dim_C \oplus \alpha_V$	0.04 (422)	0.17 (236)	0.06 (76)	0.24 (309)
$V \oplus C \oplus M \oplus R \oplus M/R$	0.27 (299)	0.09 (223)	0.04 (76)	0.21 (309)
$C \oplus Cmtys \oplus TW/V \oplus R \oplus M/R \oplus Dim_V$	0.37 (299)	0.19 (223)	0.08 (76)	0.32 (309)
$C \oplus Q \oplus Q/Cmtys \oplus M/R \oplus Dim_V \oplus \alpha_V$	0.39 (299)	0.33 (223)	0.08 (76)	0.34 (309)
$V \oplus C \oplus Q \oplus Q/Cmtys \oplus TW/V \oplus Dim_V$	0.18 (299)	0.27 (223)	0.10 (76)	0.34 (309)
$V \oplus C \oplus C/V \oplus Cmtys \oplus Q \oplus TW/V$	0.16 (299)	0.17 (223)	0.06 (76)	0.32 (309)

Table A.1: Adjusted R^2 values for the given features using ridge regression for satisfiable instances, compared to log of MapleCOMSPS' solving time. The number in parentheses indicates the number of instances that were considered in each case. The lower section considers heterogeneous sets of features across different parameter types.

Feature Set	Application	Crafted	Random	Agile
$V \oplus C \oplus C/V$	0.06 (516)	0.14 (299)	0.27 (50)	0.65 (2212)
$V \oplus C \oplus Cmtys \oplus Q \oplus Q/Cmtys$	0.08 (408)	0.22 (192)	0.42 (50)	0.70 (2212)
$V \oplus C \oplus LSR \oplus LSR/V$	0.03 (296)	0.19 (191)	0.50 (50)	0.70 (2208)
$V \oplus C \oplus TW \oplus TW/V$	0.09 (507)	0.21 (299)	0.28 (50)	0.78 (2212)
$V \oplus C \oplus Dim_V \oplus Dim_C \oplus \alpha_V$	0.08 (516)	0.21 (299)	0.39 (50)	0.68 (2177)
$V \oplus C \oplus M \oplus R \oplus M/R$	0.11 (358)	0.17 (192)	0.29 (50)	0.63 (2152)
$C \oplus Cmtys \oplus TW/V \oplus R \oplus M/R \oplus Dim_V$	0.18 (358)	0.33 (192)	0.37 (50)	0.79 (2152)
$C \oplus Q \oplus Q/Cmtys \oplus M/R \oplus Dim_V \oplus \alpha_V$	0.13 (358)	0.35 (192)	0.41 (50)	0.73 (2152)
$V \oplus C \oplus Q \oplus Q/Cmtys \oplus TW/V \oplus Dim_V$	0.14 (358)	0.32 (192)	0.52 (50)	0.81 (2152)
$V \oplus C \oplus C/V \oplus Cmtys \oplus Q \oplus TW/V$	0.10 (358)	0.26 (192)	0.35 (50)	0.82 (2152)

Table A.2: Repeated results for unsatisfiable instances.

Feature Set	Application	Crafted	Random	Agile
$V \oplus C \oplus C/V$	0.02 (1134)	0.04 (729)	0.28 (117)	0.93 (4968)
$V \oplus C \oplus Cmtys \oplus Q \oplus Q/Cmtys$	0.06 (882)	0.18 (595)	0.51 (117)	0.93 (4968)
$V \oplus C \oplus LSR \oplus LSR/V$	0.20 (696)	0.17 (412)	0.43 (116)	0.95 (3592)
$V \oplus C \oplus \#Min\_Weak \oplus Weak$	0.11 (274)	0.17 (188)	0.33 (70)	0.68 (464)
$V \oplus C \oplus Bones \oplus Bones/V$	0.11 (192)	0.43 (149)	0.38 (55)	0.77 (208)
$V \oplus C \oplus TW \oplus TW/V$	0.01 (1078)	0.06 (729)	0.47 (117)	0.95 (4968)
$V \oplus C \oplus Dim_V \oplus Dim_C \oplus \alpha_V$	0.08 (1134)	0.22 (729)	0.51 (117)	0.93 (4901)
$V \oplus C \oplus M \oplus R \oplus M/R$	0.28 (817)	0.18 (586)	0.41 (117)	0.95 (4870)
$C/V \oplus Q \oplus TW/V \oplus M/R \oplus R \oplus Dim_V$	**0.49 (817)**	0.47 (586)	0.48 (117)	0.98 (4870)
$Q \oplus TW/V \oplus R \oplus M/R \oplus Dim_V \oplus Dim_C$	0.46 (817)	**0.53 (586)**	0.41 (117)	0.97 (4870)
$V \oplus C/V \oplus Cmtys \oplus Q \oplus Dim_V \oplus M$	0.33 (817)	0.22 (586)	**0.74 (117)**	0.98 (4870)
$V \oplus C/V \oplus Q \oplus M \oplus M/R \oplus Dim_C$	0.37 (817)	0.26 (586)	0.38 (117)	**0.98 (4870)**

Table A.3: Adjusted R^2 values for the given features using linear regression, compared to log of Lingeling's solving time. The number in parentheses indicates the number of instances that were considered in each case. The lower section considers heterogeneous sets of features across different parameter types.

Feature Set	Application	Crafted	Random	Agile
$V \oplus C \oplus C/V$	0.02 (1134)	0.03 (729)	0.17 (117)	0.84 (4968)
$V \oplus C \oplus Cmtys \oplus Q \oplus Q/Cmtys$	0.03 (882)	0.09 (595)	0.26 (117)	0.79 (4968)
$V \oplus C \oplus LSR \oplus LSR/V$	0.15 (696)	0.12 (412)	0.28 (116)	0.84 (3592)
$V \oplus C \oplus \#Min\_Weak \oplus Weak$	0.01 (274)	0.10 (188)	0.16 (70)	0.19 (464)
$V \oplus C \oplus Bones \oplus Bones/V$	0.11 (192)	0.29 (149)	0.26 (55)	0.40 (208)
$V \oplus C \oplus TW \oplus TW/V$	0.02 (1078)	0.04 (729)	0.17 (117)	0.90 (4968)
$V \oplus C \oplus Dim_V \oplus Dim_C \oplus \alpha_V$	0.04 (1134)	0.14 (729)	0.30 (117)	0.83 (4901)
$V \oplus C \oplus M \oplus R \oplus M/R$	0.12 (817)	0.10 (586)	0.22 (117)	0.78 (4870)
$C \oplus C/V \oplus Q/Cmtys \oplus TW/V \oplus M/R \oplus Dim_V$	**0.23 (817)**	0.18 (586)	0.32 (117)	0.92 (4870)
$Q/Cmtys \oplus TW/V \oplus M/R \oplus Dim_V \oplus Dim_C \oplus \alpha_V$	0.16 (817)	**0.28 (586)**	0.31 (117)	0.84 (4870)
$V \oplus C \oplus Q/Cmtys \oplus TW/V \oplus M \oplus Dim_V$	0.12 (817)	0.14 (586)	**0.38 (117)**	0.91 (4870)
$V \oplus C \oplus C/V \oplus Cmtys \oplus Q \oplus TW/V$	0.09 (817)	0.11 (586)	0.26 (117)	**0.93 (4870)**

Table A.4: Repeated results using ridge regression.

Pearson	C/V	Q	Bones/V	TW/V	Weak/V	LSR/V	Merges	Merges/Res
2d-strip-packing	**0.81**	0.38	–	-0.17	–	-0.14	-0.36	-0.40
argumentation	0.59	**-0.85**	–	**0.65**	**0.95**	**0.81**	**0.93**	**0.70**
bio	0.50	-0.25	**0.96**	0.56	-0.57	0.44	-0.51	-0.43
crypto-aes	-0.03	0.21	0.52	-0.09	-0.39	-0.20	-0.08	-0.04
crypto-des	0.25	-0.06	–	-0.55	0.53	0.48	**0.70**	**0.80**
crypto-gos	0.50	0.51	–	-0.52	–	–	0.50	0.52
crypto-md5	0.47	-0.28	–	-0.10	-0.52	-0.22	0.44	0.46
crypto-sha	**-0.97**	**-0.93**	–	**0.99**	–	0.36	**0.99**	**1.00**
crypto-vmpc	**0.80**	0.58	-0.55	**0.73**	**-0.82**	–	**-0.77**	**-0.79**
diagnosis	-0.03	0.14	0.03	-0.06	0.29	-0.10	-0.15	-0.07
hardware-bmc	-0.10	0.17	–	-0.13	–	0.53	-0.29	0.23
hardware-bmc-ibm	-0.11	0.20	–	-0.24	–	–	-0.41	-0.49
hardware-cec	0.12	0.05	–	-0.29	–	-0.57	**0.64**	**0.67**
hardware-manolios	0.31	0.47	–	**-0.66**	–	-0.41	-0.41	**-0.66**
hardware-velev	0.06	0.56	–	-0.29	–	-0.33	–	–
planning	-0.32	-0.47	–	0.04	-0.20	0.18	-0.46	-0.42
scheduling	0.11	0.18	–	-0.23	-0.16	-0.37	-0.09	-0.09
scheduling-pesp	-0.20	-0.14	–	0.16	–	-0.14	-0.52	-0.55
software-bit-verif	-0.15	0.41	–	-0.16	–	0.22	-0.20	-0.29
termination	0.30	-0.20	0.15	0.26	-0.14	0.58	0.08	0.07

Spearman	C/V	Q	Bones/V	TW/V	Weak/V	LSR/V	Merges	Merges/Res
2d-strip-packing	**0.64**	**0.69**	–	-0.20	–	0.27	**-0.88**	**-0.77**
argumentation	**0.68**	**-0.79**	–	**0.69**	-0.04	0.31	**0.79**	**0.74**
bio	0.24	-0.25	**0.79**	0.22	-0.56	0.10	-0.36	-0.57
crypto-aes	0.11	0.27	0.39	-0.40	-0.44	-0.25	-0.39	-0.22
crypto-des	0.20	-0.11	–	**-0.65**	0.53	**0.69**	**0.72**	**0.85**
crypto-gos	0.16	0.43	–	-0.07	–	–	0.09	0.41
crypto-md5	0.52	-0.10	–	-0.07	0.20	-0.34	0.35	0.50
crypto-sha	**-0.77**	**-0.78**	–	**0.77**	–	0.46	**0.77**	**0.76**
crypto-vmpc	**0.84**	0.58	**-0.62**	**0.78**	**-0.82**	–	**-0.84**	**-0.84**
diagnosis	0.03	0.34	0.27	-0.15	0.31	0.19	-0.38	-0.14
hardware-bmc	0.09	0.17	–	-0.21	–	**0.93**	**-0.62**	-0.41
hardware-bmc-ibm	-0.00	0.57	–	-0.17	–	–	-0.21	-0.28
hardware-cec	0.49	0.29	–	0.00	–	-0.38	0.02	0.07
hardware-manolios	0.48	**0.70**	–	**-0.91**	–	-0.40	-0.82	**-0.82**
hardware-velev	0.02	0.48	–	-0.39	–	0.17	–	–
planning	-0.38	-0.43	–	-0.02	-0.53	0.21	-0.34	-0.31
scheduling	-0.12	0.32	–	-0.20	0.03	-0.41	0.25	0.26
scheduling-pesp	-0.27	-0.03	–	0.06	–	-0.18	-0.54	**-0.66**
software-bit-verif	-0.21	0.56	–	-0.10	–	0.27	-0.60	-0.27
termination	0.26	-0.02	0.05	0.05	**-0.65**	0.58	0.09	0.31

Table A.5: Pearson (top) and Spearman (bottom) correlations between measures and Lingeling solving time. Omits entries with less than 10 data points.

Dep. Variable:	log(MapleCOMSPS_time)	**R-squared:**	0.365
Model:	OLS	**Adj. R-squared:**	0.312
Method:	Least Squares	**F-statistic:**	6.930
Date:	Wed, 28 Mar 2018	**Prob (F-statistic):**	7.33e-43
Time:	13:53:38	**Log-Likelihood:**	-2032.6
No. Observations:	823	**AIC:**	4193.
Df Residuals:	759	**BIC:**	4495.
Df Model:	63		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
Intercept	5.0082	0.640	7.830	0.000	3.753	6.264
C/V	7.1926	2.111	3.407	0.001	3.048	11.337
Q	-0.6574	0.942	-0.698	0.486	-2.507	1.192
TW/V	0.5078	0.864	0.588	0.557	-1.188	2.204
M/R	0.7774	0.559	1.391	0.165	-0.320	1.875
R	-0.6117	2.005	-0.305	0.760	-4.547	3.324
Dim$_{\mathbf{VIG}}$	1.1800	1.020	1.157	0.248	-0.822	3.182
C/V:Q	-3.5286	3.029	-1.165	0.244	-9.474	2.417
C/V:TW/V	5.8359	2.677	2.180	0.030	0.581	11.090
C/V:M/R	2.8615	2.296	1.246	0.213	-1.646	7.370
C/V:R	7.0877	5.730	1.237	0.216	-4.161	18.336
C/V:Dim$_{\mathbf{VIG}}$	5.5440	2.926	1.895	0.058	-0.199	11.287
Q:TW/V	3.3855	1.615	2.096	0.036	0.215	6.556
Q:M/R	-2.3947	1.380	-1.735	0.083	-5.104	0.315
Q:R	0.3264	1.586	0.206	0.837	-2.786	3.439
Q:Dim$_{\mathbf{VIG}}$	-0.7065	1.257	-0.562	0.574	-3.174	1.761
TW/V:M/R	0.3742	0.971	0.386	0.700	-1.531	2.280
TW/V:R	0.3926	2.167	0.181	0.856	-3.861	4.646
TW/V:Dim$_{\mathbf{VIG}}$	1.6676	1.638	1.018	0.309	-1.549	4.884
M/R:R	3.4688	1.509	2.298	0.022	0.506	6.432
M/R:Dim$_{\mathbf{VIG}}$	-3.0861	1.410	-2.189	0.029	-5.854	-0.319
R:Dim$_{\mathbf{VIG}}$	7.8470	2.535	3.096	0.002	2.871	12.823
C/V:Q:TW/V	-3.9710	3.530	-1.125	0.261	-10.901	2.959
C/V:Q:M/R	-12.4658	4.596	-2.712	0.007	-21.488	-3.444
C/V:Q:R	10.3432	3.960	2.612	0.009	2.569	18.117
C/V:Q:Dim$_{\mathbf{VIG}}$	-8.4177	3.370	-2.498	0.013	-15.033	-1.803
C/V:TW/V:M/R	10.0316	4.027	2.491	0.013	2.126	17.938
C/V:TW/V:R	-2.0699	6.139	-0.337	0.736	-14.122	9.982
C/V:TW/V:Dim$_{\mathbf{VIG}}$	7.0906	4.161	1.704	0.089	-1.078	15.259
C/V:M/R:R	-10.0587	5.588	-1.800	0.072	-21.029	0.911
C/V:M/R:Dim$_{\mathbf{VIG}}$	4.8363	3.960	1.221	0.222	-2.937	12.609
C/V:R:Dim$_{\mathbf{VIG}}$	10.1645	6.652	1.528	0.127	-2.894	23.223
Q:TW/V:M/R	3.3889	2.776	1.221	0.222	-2.060	8.838
Q:TW/V:R	-1.3517	1.069	-1.264	0.207	-3.451	0.748
Q:TW/V:Dim$_{\mathbf{VIG}}$	5.1535	1.290	3.995	0.000	2.621	7.686
Q:M/R:R	-2.4894	1.904	-1.308	0.191	-6.227	1.248
Q:M/R:Dim$_{\mathbf{VIG}}$	-5.0984	2.115	-2.411	0.016	-9.250	-0.947
Q:R:Dim$_{\mathbf{VIG}}$	0.7866	1.453	0.541	0.588	-2.066	3.639
TW/V:M/R:R	-2.3149	1.613	-1.435	0.152	-5.481	0.851
TW/V:M/R:Dim$_{\mathbf{VIG}}$	-0.6613	2.501	-0.264	0.792	-5.571	4.248
TW/V:R:Dim$_{\mathbf{VIG}}$	-1.8055	2.095	-0.862	0.389	-5.918	2.307
M/R:R:Dim$_{\mathbf{VIG}}$	0.5126	2.494	0.206	0.837	-4.383	5.408
C/V:Q:TW/V:M/R	-2.9167	6.093	-0.479	0.632	-14.879	9.045
C/V:Q:TW/V:R	-2.7630	3.319	-0.833	0.405	-9.278	3.752
C/V:Q:TW/V:Dim$_{\mathbf{VIG}}$	4.9389	1.659	2.976	0.003	1.681	8.196
C/V:Q:M/R:R	2.3426	3.534	0.663	0.508	-4.596	9.281
C/V:Q:M/R:Dim$_{\mathbf{VIG}}$	-21.7895	5.625	-3.874	0.000	-32.831	-10.748
C/V:Q:R:Dim$_{\mathbf{VIG}}$	1.9698	3.432	0.574	0.566	-4.768	8.708
C/V:TW/V:M/R:R	-0.2357	5.676	-0.042	0.967	-11.379	10.907
C/V:TW/V:M/R:Dim$_{\mathbf{VIG}}$	10.8523	7.183	1.511	0.131	-3.249	24.954
C/V:TW/V:R:Dim$_{\mathbf{VIG}}$	-10.6888	5.089	-2.101	0.036	-20.678	-0.699
C/V:M/R:R:Dim$_{\mathbf{VIG}}$	0.1316	5.001	0.026	0.979	-9.685	9.948
Q:TW/V:M/R:R	0.0388	1.397	0.028	0.978	-2.704	2.781
Q:TW/V:M/R:Dim$_{\mathbf{VIG}}$	7.3008	1.865	3.914	0.000	3.639	10.962
Q:TW/V:R:Dim$_{\mathbf{VIG}}$	-0.3096	0.842	-0.368	0.713	-1.963	1.344
Q:M/R:R:Dim$_{\mathbf{VIG}}$	-1.2530	1.967	-0.637	0.524	-5.115	2.609
TW/V:M/R:R:Dim$_{\mathbf{VIG}}$	-1.7819	2.515	-0.709	0.479	-6.719	3.155
C/V:Q:TW/V:M/R:R	1.9072	2.150	0.887	0.375	-2.314	6.129
C/V:Q:TW/V:M/R:Dim$_{\mathbf{VIG}}$	9.3722	2.369	3.957	0.000	4.723	14.022
C/V:Q:TW/V:R:Dim$_{\mathbf{VIG}}$	-1.8732	1.554	-1.205	0.229	-4.925	1.178
C/V:Q:M/R:R:Dim$_{\mathbf{VIG}}$	7.9418	3.198	2.484	0.013	1.664	14.219
C/V:TW/V:M/R:R:Dim$_{\mathbf{VIG}}$	-2.2511	4.247	-0.530	0.596	-10.588	6.086
Q:TW/V:M/R:R:Dim$_{\mathbf{VIG}}$	-2.1006	1.347	-1.559	0.119	-4.745	0.544
C/V:Q:TW/V:M/R:R:Dim$_{\mathbf{VIG}}$	-3.3675	1.709	-1.970	0.049	-6.723	-0.012

Table A.6: Coefficients for best found model for application instances.

Dep. Variable:	log(MapleCOMSPS_time)	**R-squared:**	0.608
Model:	OLS	**Adj. R-squared:**	0.562
Method:	Least Squares	**F-statistic:**	13.28
Date:	Wed, 28 Mar 2018	**Prob (F-statistic):**	2.63e-75
Time:	13:53:55	**Log-Likelihood:**	-1415.8
No. Observations:	604	**AIC:**	2960.
Df Residuals:	540	**BIC:**	3241.
Df Model:	63		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
Intercept	3.6524	0.637	5.736	0.000	2.402	4.903
Q	-2.1729	0.912	-2.382	0.018	-3.964	-0.381
Q/Cmtys	-3.9696	0.788	-5.036	0.000	-5.518	-2.421
TW/V	-2.9561	0.828	-3.571	0.000	-4.582	-1.330
M/R	-2.3963	1.101	-2.176	0.030	-4.559	-0.233
Dim_{VIG}	-1.4365	1.015	-1.416	0.157	-3.430	0.557
Dim_{CVIG}	1.4747	1.009	1.462	0.144	-0.507	3.457
Q:Q/Cmtys	-4.2913	1.147	-3.740	0.000	-6.545	-2.037
Q:TW/V	0.3525	0.945	0.373	0.709	-1.504	2.209
Q:M/R	-1.1392	1.675	-0.680	0.497	-4.430	2.152
Q:Dim_{VIG}	-3.9890	1.462	-2.728	0.007	-6.862	-1.116
Q:Dim_{CVIG}	4.7651	1.319	3.612	0.000	2.174	7.357
Q/Cmtys:TW/V	-2.4263	0.900	-2.695	0.007	-4.195	-0.658
Q/Cmtys:M/R	-7.0206	1.312	-5.350	0.000	-9.598	-4.443
Q/Cmtys:Dim_{VIG}	-1.2185	0.994	-1.226	0.221	-3.171	0.734
Q/Cmtys:Dim_{CVIG}	-0.2478	1.302	-0.190	0.849	-2.805	2.309
TW/V:M/R	-2.7631	1.444	-1.913	0.056	-5.600	0.074
TW/V:Dim_{VIG}	-3.5089	0.983	-3.568	0.000	-5.441	-1.577
TW/V:Dim_{CVIG}	5.6229	0.957	5.873	0.000	3.742	7.504
M/R:Dim_{VIG}	1.3015	1.814	0.717	0.473	-2.262	4.865
M/R:Dim_{CVIG}	-0.6769	1.666	-0.406	0.685	-3.950	2.596
Dim_{VIG}:Dim_{CVIG}	1.3038	0.641	2.033	0.043	0.044	2.564
Q:Q/Cmtys:TW/V	-2.4838	1.134	-2.190	0.029	-4.712	-0.256
Q:Q/Cmtys:M/R	-9.3369	1.849	-5.049	0.000	-12.970	-5.704
Q:Q/Cmtys:Dim_{VIG}	-3.4448	1.911	-1.803	0.072	-7.199	0.309
Q:Q/Cmtys:Dim_{CVIG}	4.0473	2.133	1.897	0.058	-0.143	8.238
Q:TW/V:M/R	1.5347	1.515	1.013	0.311	-1.441	4.510
Q:TW/V:Dim_{VIG}	-0.9245	1.332	-0.694	0.488	-3.541	1.693
Q:TW/V:Dim_{CVIG}	-1.1701	1.099	-1.065	0.287	-3.328	0.988
Q:M/R:Dim_{VIG}	-6.0967	2.640	-2.309	0.021	-11.283	-0.911
Q:M/R:Dim_{CVIG}	7.4145	2.438	3.042	0.002	2.626	12.203
Q:Dim_{VIG}:Dim_{CVIG}	1.7028	0.810	2.102	0.036	0.112	3.294
Q/Cmtys:TW/V:M/R	-7.8250	1.624	-4.819	0.000	-11.015	-4.635
Q/Cmtys:TW/V:Dim_{VIG}	-6.6613	1.189	-5.603	0.000	-8.997	-4.326
Q/Cmtys:TW/V:Dim_{CVIG}	5.5079	1.445	3.811	0.000	2.669	8.347
Q/Cmtys:M/R:Dim_{VIG}	-3.6912	1.513	-2.440	0.015	-6.663	-0.719
Q/Cmtys:M/R:Dim_{CVIG}	-1.1475	2.190	-0.524	0.600	-5.449	3.154
Q/Cmtys:Dim_{VIG}:Dim_{CVIG}	1.7641	0.994	1.775	0.076	-0.188	3.716
TW/V:M/R:Dim_{VIG}	-2.7635	1.766	-1.564	0.118	-6.234	0.707
TW/V:M/R:Dim_{CVIG}	7.8079	1.800	4.338	0.000	4.272	11.344
TW/V:Dim_{VIG}:Dim_{CVIG}	-0.1617	0.709	-0.228	0.820	-1.554	1.231
M/R:Dim_{VIG}:Dim_{CVIG}	-0.3330	0.939	-0.355	0.723	-2.178	1.512
Q:Q/Cmtys:TW/V:M/R	-4.6870	1.932	-2.426	0.016	-8.482	-0.892
Q:Q/Cmtys:TW/V:Dim_{VIG}	5.5356	1.680	3.296	0.001	2.236	8.835
Q:Q/Cmtys:TW/V:Dim_{CVIG}	-3.9253	1.721	-2.281	0.023	-7.306	-0.545
Q:Q/Cmtys:M/R:Dim_{VIG}	-12.6068	3.152	-4.000	0.000	-18.798	-6.415
Q:Q/Cmtys:M/R:Dim_{CVIG}	9.9667	3.557	2.802	0.005	2.979	16.954
Q:Q/Cmtys:Dim_{VIG}:Dim_{CVIG}	1.9627	1.933	1.016	0.310	-1.833	5.759
Q:TW/V:M/R:Dim_{VIG}	1.6740	2.277	0.735	0.463	-2.800	6.148
Q:TW/V:M/R:Dim_{CVIG}	-4.9751	1.813	-2.744	0.006	-8.536	-1.414
Q:TW/V:Dim_{VIG}:Dim_{CVIG}	0.6247	0.653	0.957	0.339	-0.657	1.907
Q:M/R:Dim_{VIG}:Dim_{CVIG}	0.3896	1.235	0.315	0.753	-2.037	2.817
Q/Cmtys:TW/V:M/R:Dim_{VIG}	-13.2494	2.087	-6.349	0.000	-17.349	-9.150
Q/Cmtys:TW/V:M/R:Dim_{CVIG}	7.9367	2.654	2.990	0.003	2.723	13.151
Q/Cmtys:TW/V:Dim_{VIG}:Dim_{CVIG}	4.7943	0.965	4.969	0.000	2.899	6.690
Q/Cmtys:M/R:Dim_{VIG}:Dim_{CVIG}	5.1495	1.519	3.389	0.001	2.165	8.134
TW/V:M/R:Dim_{VIG}:Dim_{CVIG}	0.4022	1.227	0.328	0.743	-2.009	2.813
Q:Q/Cmtys:TW/V:M/R:Dim_{VIG}	3.6523	2.854	1.280	0.201	-1.954	9.258
Q:Q/Cmtys:TW/V:M/R:Dim_{CVIG}	-8.6876	2.854	-3.044	0.002	-14.295	-3.080
Q:Q/Cmtys:TW/V:Dim_{VIG}:Dim_{CVIG}	-0.7901	1.443	-0.548	0.584	-3.624	2.044
Q:Q/Cmtys:M/R:Dim_{VIG}:Dim_{CVIG}	6.1376	2.843	2.159	0.031	0.554	11.722
Q:TW/V:M/R:Dim_{VIG}:Dim_{CVIG}	2.8662	1.097	2.612	0.009	0.710	5.022
Q/Cmtys:TW/V:M/R:Dim_{VIG}:Dim_{CVIG}	7.3598	1.651	4.458	0.000	4.117	10.603
Q:Q/Cmtys:TW/V:M/R:Dim_{VIG}:Dim_{CVIG}	-0.1215	2.450	-0.050	0.960	-4.935	4.692

Table A.7: Coefficients for best found model for crafted instances.

Dep. Variable:	log(MapleCOMSPS_time)	**R-squared:**	0.830
Model:	OLS	**Adj. R-squared:**	0.657
Method:	Least Squares	**F-statistic:**	4.797
Date:	Wed, 28 Mar 2018	**Prob (F-statistic):**	2.00e-09
Time:	13:54:07	**Log-Likelihood:**	-115.94
No. Observations:	126	**AIC:**	359.9
Df Residuals:	62	**BIC:**	541.4
Df Model:	63		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
Intercept	-2.044e+04	4311.465	-4.740	0.000	-2.91e+04	-1.18e+04
V	-2.764e+04	6143.937	-4.499	0.000	-3.99e+04	-1.54e+04
Cmtys	-2637.8133	6036.581	-0.437	0.664	-1.47e+04	9429.139
Q	2423.2279	6390.650	0.379	0.706	-1.04e+04	1.52e+04
R	-3.24e+04	6814.831	-4.755	0.000	-4.6e+04	-1.88e+04
Dim$_{\mathbf{VIG}}$	-1.039e+04	2845.355	-3.653	0.001	-1.61e+04	-4704.884
$\alpha_{\mathbf{V}}$	-3.161e+04	9121.150	-3.466	0.001	-4.98e+04	-1.34e+04
V:Cmtys	-2662.3995	8650.814	-0.308	0.759	-2e+04	1.46e+04
V:Q	4460.0976	9252.170	0.482	0.631	-1.4e+04	2.3e+04
V:R	-4.131e+04	9267.191	-4.458	0.000	-5.98e+04	-2.28e+04
V:Dim$_{\mathbf{VIG}}$	-1.714e+04	4444.740	-3.856	0.000	-2.6e+04	-8256.211
V:$\alpha_{\mathbf{V}}$	-4.19e+04	1.3e+04	-3.222	0.002	-6.79e+04	-1.59e+04
Cmtys:Q	-5657.5898	3195.878	-1.770	0.082	-1.2e+04	730.878
Cmtys:R	-4730.0421	9519.290	-0.497	0.621	-2.38e+04	1.43e+04
Cmtys:Dim$_{\mathbf{VIG}}$	-1.54e+04	5401.631	-2.852	0.006	-2.62e+04	-4606.815
Cmtys:$\alpha_{\mathbf{V}}$	-4946.3984	1.19e+04	-0.415	0.680	-2.88e+04	1.89e+04
Q:R	3725.5650	1.01e+04	0.368	0.714	-1.65e+04	2.4e+04
Q:Dim$_{\mathbf{VIG}}$	-1.531e+04	3796.251	-4.033	0.000	-2.29e+04	-7721.575
Q:$\alpha_{\mathbf{V}}$	-1526.0941	1.16e+04	-0.132	0.896	-2.47e+04	2.16e+04
R:Dim$_{\mathbf{VIG}}$	-1.696e+04	4636.362	-3.658	0.001	-2.62e+04	-7693.892
R:$\alpha_{\mathbf{V}}$	-5.026e+04	1.45e+04	-3.478	0.001	-7.91e+04	-2.14e+04
Dim$_{\mathbf{VIG}}$:$\alpha_{\mathbf{V}}$	-1.729e+04	5459.793	-3.166	0.002	-2.82e+04	-6371.930
V:Cmtys:Q	-6886.1033	4310.672	-1.597	0.115	-1.55e+04	1730.808
V:Cmtys:R	-4042.8705	1.31e+04	-0.309	0.759	-3.02e+04	2.22e+04
V:Cmtys:Dim$_{\mathbf{VIG}}$	-2.387e+04	7941.489	-3.006	0.004	-3.97e+04	-7994.906
V:Cmtys:$\alpha_{\mathbf{V}}$	-6086.2173	1.72e+04	-0.354	0.724	-4.04e+04	2.82e+04
V:Q:R	6527.1126	1.4e+04	0.465	0.644	-2.15e+04	3.46e+04
V:Q:Dim$_{\mathbf{VIG}}$	-2.348e+04	5611.360	-4.184	0.000	-3.47e+04	-1.23e+04
V:Q:$\alpha_{\mathbf{V}}$	1398.4545	1.68e+04	0.083	0.934	-3.23e+04	3.51e+04
V:R:Dim$_{\mathbf{VIG}}$	-2.746e+04	7094.156	-3.871	0.000	-4.16e+04	-1.33e+04
V:R:$\alpha_{\mathbf{V}}$	-6.244e+04	1.96e+04	-3.180	0.002	-1.02e+05	-2.32e+04
V:Dim$_{\mathbf{VIG}}$:$\alpha_{\mathbf{V}}$	-2.857e+04	8344.362	-3.424	0.001	-4.52e+04	-1.19e+04
Cmtys:Q:R	-8456.5008	4990.173	-1.695	0.095	-1.84e+04	1518.714
Cmtys:Q:Dim$_{\mathbf{VIG}}$	-1.229e+04	4462.134	-2.753	0.008	-2.12e+04	-3366.476
Cmtys:Q:$\alpha_{\mathbf{V}}$	-8394.7258	7202.695	-1.165	0.248	-2.28e+04	6003.256
Cmtys:R:Dim$_{\mathbf{VIG}}$	-2.526e+04	8790.431	-2.874	0.006	-4.28e+04	-7687.540
Cmtys:R:$\alpha_{\mathbf{V}}$	-8820.2403	1.88e+04	-0.469	0.641	-4.64e+04	2.88e+04
Cmtys:Dim$_{\mathbf{VIG}}$:$\alpha_{\mathbf{V}}$	-2.999e+04	9304.069	-3.223	0.002	-4.86e+04	-1.14e+04
Q:R:Dim$_{\mathbf{VIG}}$	-2.377e+04	6028.665	-3.943	0.000	-3.58e+04	-1.17e+04
Q:R:$\alpha_{\mathbf{V}}$	-2395.5332	1.83e+04	-0.131	0.897	-3.91e+04	3.43e+04
Q:Dim$_{\mathbf{VIG}}$:$\alpha_{\mathbf{V}}$	-2.495e+04	8452.988	-2.952	0.004	-4.19e+04	-8056.631
R:Dim$_{\mathbf{VIG}}$:$\alpha_{\mathbf{V}}$	-2.861e+04	8848.827	-3.234	0.002	-4.63e+04	-1.09e+04
V:Cmtys:Q:R	-1.046e+04	6561.650	-1.593	0.116	-2.36e+04	2661.451
V:Cmtys:Q:Dim$_{\mathbf{VIG}}$	-1.934e+04	6635.758	-2.915	0.005	-3.26e+04	-6076.555
V:Cmtys:Q:$\alpha_{\mathbf{V}}$	-9592.3874	9994.049	-0.960	0.341	-2.96e+04	1.04e+04
V:Cmtys:R:Dim$_{\mathbf{VIG}}$	-3.779e+04	1.24e+04	-3.055	0.003	-6.25e+04	-1.31e+04
V:Cmtys:R:$\alpha_{\mathbf{V}}$	-9160.6573	2.6e+04	-0.352	0.726	-6.11e+04	4.28e+04
V:Cmtys:Dim$_{\mathbf{VIG}}$:$\alpha_{\mathbf{V}}$	-4.532e+04	1.37e+04	-3.307	0.002	-7.27e+04	-1.79e+04
V:Q:R:Dim$_{\mathbf{VIG}}$	-3.439e+04	8340.816	-4.124	0.000	-5.11e+04	-1.77e+04
V:Q:R:$\alpha_{\mathbf{V}}$	1648.6557	2.56e+04	0.064	0.949	-4.95e+04	5.28e+04
V:Q:Dim$_{\mathbf{VIG}}$:$\alpha_{\mathbf{V}}$	-3.925e+04	1.24e+04	-3.165	0.002	-6.4e+04	-1.45e+04
V:R:Dim$_{\mathbf{VIG}}$:$\alpha_{\mathbf{V}}$	-4.643e+04	1.32e+04	-3.524	0.001	-7.28e+04	-2.01e+04
Cmtys:Q:R:Dim$_{\mathbf{VIG}}$	-1.88e+04	6877.872	-2.733	0.008	-3.25e+04	-5051.971
Cmtys:Q:R:$\alpha_{\mathbf{V}}$	-1.244e+04	1.13e+04	-1.101	0.275	-3.5e+04	1.01e+04
Cmtys:Q:Dim$_{\mathbf{VIG}}$:$\alpha_{\mathbf{V}}$	-2.642e+04	7805.998	-3.385	0.001	-4.2e+04	-1.08e+04
Cmtys:R:Dim$_{\mathbf{VIG}}$:$\alpha_{\mathbf{V}}$	-4.889e+04	1.51e+04	-3.232	0.002	-7.91e+04	-1.86e+04
Q:R:Dim$_{\mathbf{VIG}}$:$\alpha_{\mathbf{V}}$	-3.813e+04	1.33e+04	-2.858	0.006	-6.48e+04	-1.15e+04
V:Cmtys:Q:R:Dim$_{\mathbf{VIG}}$	-2.787e+04	9729.329	-2.865	0.006	-4.73e+04	-8423.152
V:Cmtys:Q:R:$\alpha_{\mathbf{V}}$	-1.462e+04	1.52e+04	-0.964	0.339	-4.5e+04	1.57e+04
V:Cmtys:Q:Dim$_{\mathbf{VIG}}$:$\alpha_{\mathbf{V}}$	-4.087e+04	1.13e+04	-3.604	0.001	-6.35e+04	-1.82e+04
V:Cmtys:R:Dim$_{\mathbf{VIG}}$:$\alpha_{\mathbf{V}}$	-7.084e+04	2.13e+04	-3.325	0.001	-1.13e+05	-2.82e+04
V:Q:R:Dim$_{\mathbf{VIG}}$:$\alpha_{\mathbf{V}}$	-5.714e+04	1.85e+04	-3.096	0.003	-9.4e+04	-2.03e+04
Cmtys:Q:R:Dim$_{\mathbf{VIG}}$:$\alpha_{\mathbf{V}}$	-4.1e+04	1.22e+04	-3.367	0.001	-6.53e+04	-1.67e+04
V:Cmtys:Q:R:Dim$_{\mathbf{VIG}}$:$\alpha_{\mathbf{V}}$	-5.984e+04	1.68e+04	-3.559	0.001	-9.34e+04	-2.62e+04

Table A.8: Coefficients for best found model for random instances.

Dep. Variable:	log(MapleCOMSPS_time)	**R-squared:**	0.961
Model:	OLS	**Adj. R-squared:**	0.961
Method:	Least Squares	**F-statistic:**	1903.
Date:	Wed, 28 Mar 2018	**Prob (F-statistic):**	0.00
Time:	13:54:27	**Log-Likelihood:**	-4639.0
No. Observations:	4870	**AIC:**	9406.
Df Residuals:	4806	**BIC:**	9821.
Df Model:	63		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
Intercept	34.5233	1.156	29.873	0.000	32.258	36.789
V	28.8710	1.011	28.547	0.000	26.888	30.854
C/V	29.3222	1.284	22.844	0.000	26.806	31.839
Q	-16.6643	0.932	-17.885	0.000	-18.491	-14.838
TW/V	13.0318	1.342	9.712	0.000	10.401	15.663
M	196.2460	7.138	27.495	0.000	182.253	210.239
Dim$_{\text{CVIG}}$	-18.3519	0.888	-20.657	0.000	-20.094	-16.610
V:C/V	26.4450	1.127	23.469	0.000	24.236	28.654
V:Q	-13.8379	0.792	-17.476	0.000	-15.390	-12.286
V:TW/V	11.7906	1.147	10.277	0.000	9.541	14.040
V:M	130.3730	4.600	28.342	0.000	121.355	139.391
V:Dim$_{\text{CVIG}}$	-15.9007	0.765	-20.797	0.000	-17.400	-14.402
C/V:Q	7.2377	0.890	8.129	0.000	5.492	8.983
C/V:TW/V	-3.1250	0.389	-8.042	0.000	-3.887	-2.363
C/V:M	178.6771	7.896	22.628	0.000	163.197	194.157
C/V:Dim$_{\text{CVIG}}$	-13.4137	0.824	-16.281	0.000	-15.029	-11.799
Q:TW/V	30.4944	0.931	32.764	0.000	28.670	32.319
Q:M	-106.3497	5.786	-18.379	0.000	-117.694	-95.006
Q:Dim$_{\text{CVIG}}$	1.9093	0.683	2.797	0.005	0.571	3.248
TW/V:M	83.6501	8.214	10.184	0.000	67.547	99.753
TW/V:Dim$_{\text{CVIG}}$	-5.6574	1.189	-4.756	0.000	-7.989	-3.326
M:Dim$_{\text{CVIG}}$	-117.5625	5.408	-21.740	0.000	-128.164	-106.961
V:C/V:Q	7.2789	0.776	9.385	0.000	5.758	8.800
V:C/V:TW/V	-3.1542	0.335	-9.411	0.000	-3.811	-2.497
V:C/V:M	114.4664	5.031	22.751	0.000	104.603	124.330
V:C/V:Dim$_{\text{CVIG}}$	-11.7793	0.712	-16.555	0.000	-13.174	-10.384
V:Q:TW/V	25.3693	0.769	32.996	0.000	23.862	26.877
V:Q:M	-63.4485	3.569	-17.777	0.000	-70.446	-56.451
V:Q:Dim$_{\text{CVIG}}$	1.3735	0.577	2.379	0.017	0.242	2.505
V:TW/V:M	51.8590	5.204	9.964	0.000	41.656	62.062
V:TW/V:Dim$_{\text{CVIG}}$	-5.2254	1.004	-5.207	0.000	-7.193	-3.258
V:M:Dim$_{\text{CVIG}}$	-73.7585	3.412	-21.618	0.000	-80.447	-67.070
C/V:Q:TW/V	4.9761	0.523	9.517	0.000	3.951	6.001
C/V:Q:M	56.5226	5.724	9.875	0.000	45.301	67.744
C/V:Q:Dim$_{\text{CVIG}}$	1.8702	0.562	3.327	0.001	0.768	2.972
C/V:TW/V:M	-18.9932	2.314	-8.208	0.000	-23.530	-14.457
C/V:TW/V:Dim$_{\text{CVIG}}$	1.8945	0.300	6.310	0.000	1.306	2.483
C/V:M:Dim$_{\text{CVIG}}$	-79.5336	4.990	-15.939	0.000	-89.316	-69.751
Q:TW/V:M	183.4344	5.827	31.482	0.000	172.012	194.857
Q:TW/V:Dim$_{\text{CVIG}}$	-0.6468	0.747	-0.866	0.387	-2.111	0.818
Q:M:Dim$_{\text{CVIG}}$	10.0300	4.283	2.342	0.019	1.634	18.426
TW/V:M:Dim$_{\text{CVIG}}$	-35.4234	7.245	-4.889	0.000	-49.627	-21.220
V:C/V:Q:TW/V	4.2193	0.448	9.424	0.000	3.342	5.097
V:C/V:Q:M	36.8879	3.635	10.149	0.000	29.763	44.013
V:C/V:Q:Dim$_{\text{CVIG}}$	0.8428	0.482	1.747	0.081	-0.103	1.789
V:C/V:TW/V:M	-12.2890	1.478	-8.317	0.000	-15.186	-9.392
V:C/V:TW/V:Dim$_{\text{CVIG}}$	1.6428	0.248	6.619	0.000	1.156	2.129
V:C/V:M:Dim$_{\text{CVIG}}$	-49.7462	3.182	-15.631	0.000	-55.985	-43.507
V:Q:TW/V:M	114.4683	3.586	31.920	0.000	107.438	121.499
V:Q:TW/V:Dim$_{\text{CVIG}}$	-1.6760	0.598	-2.804	0.005	-2.848	-0.504
V:Q:M:Dim$_{\text{CVIG}}$	7.6337	2.635	2.897	0.004	2.468	12.800
V:TW/V:M:Dim$_{\text{CVIG}}$	-21.4927	4.459	-4.820	0.000	-30.234	-12.751
C/V:Q:TW/V:M	31.4922	2.840	11.089	0.000	25.924	37.060
C/V:Q:TW/V:Dim$_{\text{CVIG}}$	-0.0167	0.314	-0.053	0.958	-0.632	0.598
C/V:Q:M:Dim$_{\text{CVIG}}$	2.6130	3.551	0.736	0.462	-4.349	9.575
C/V:TW/V:M:Dim$_{\text{CVIG}}$	7.6758	1.785	4.299	0.000	4.176	11.176
Q:TW/V:M:Dim$_{\text{CVIG}}$	-13.4918	4.578	-2.947	0.003	-22.467	-4.516
V:C/V:Q:TW/V:M	19.3458	1.797	10.767	0.000	15.823	22.868
V:C/V:Q:TW/V:Dim$_{\text{CVIG}}$	0.0691	0.264	0.261	0.794	-0.449	0.588
V:C/V:Q:M:Dim$_{\text{CVIG}}$	2.2624	2.279	0.993	0.321	-2.205	6.730
V:C/V:TW/V:M:Dim$_{\text{CVIG}}$	4.8482	1.104	4.392	0.000	2.684	7.012
V:Q:TW/V:M:Dim$_{\text{CVIG}}$	-8.8265	2.859	-3.087	0.002	-14.431	-3.221
C/V:Q:TW/V:M:Dim$_{\text{CVIG}}$	1.3627	1.845	0.739	0.460	-2.254	4.979
V:C/V:Q:TW/V:M:Dim$_{\text{CVIG}}$	0.6254	1.175	0.532	0.595	-1.679	2.930

Table A.9: Coefficients for best found model for agile instances.

Appendix B

List of Application Instances Used in Lens Studies

1. sc09-app/ACG-10-5p0.cnf
2. sc09-app/aloul-chnl11-13.cnf
3. sc09-app/AProVE09-07.cnf
4. sc09-app/AProVE09-11.cnf
5. sc09-app/AProVE09-17.cnf
6. sc09-app/cmu-bmc-longmult15.cnf
7. sc09-app/gss-15-s100.cnf
8. sc09-app/gss-17-s100.cnf
9. sc09-app/gus-md5-04.cnf
10. sc09-app/gus-md5-06.cnf
11. sc09-app/gus-md5-07.cnf
12. sc09-app/manol-pipe-c6nidw_i.cnf
13. sc09-app/manol-pipe-g10id.cnf
14. sc09-app/mizh-sha0-35-3.cnf
15. sc09-app/post-cbmc-aes-ele-noholes.cnf
16. sc09-app/q_query_3_l39_lambda.cnf
17. sc09-app/q_query_3_l40_lambda.cnf
18. sc09-app/q_query_3_l42_lambda.cnf
19. sc09-app/q_query_3_l43_lambda.cnf
20. sc09-app/q_query_3_l44_lambda.cnf
21. sc09-app/q_query_3_L60_coli.sat.cnf
22. sc09-app/schup-l2s-abp4-1-k31.cnf
23. sc09-app/schup-l2s-motst-2-k315.cnf
24. sc09-app/simon-s03-w08-15.cnf
25. sc09-app/smulo016.cnf

26. sc09-app/total-5-13-u.cnf
27. sc09-app/UCG-10-5p0.cnf
28. sc09-app/UR-10-5p1.cnf
29. sc09-app/vmpc_24.cnf
30. sc09-app/vmpc_28.cnf
31. sc09-app/vmpc_34.cnf
32. sc11-app/AProVE11-07.cnf
33. sc11-app/AProVE11-10.cnf
34. sc11-app/blocks-4-ipc5-h21-unknown.cnf
35. sc11-app/E02F17.cnf
36. sc11-app/E05X15.cnf
37. sc11-app/gss-16-s100.cnf
38. sc11-app/homer16.shuffled.cnf
39. sc11-app/ibm-2002-21r-k95.cnf
40. sc11-app/manol-pipe-c6bidw_i.cnf
41. sc11-app/myciel6-tr.used-as.sat04-320.cnf
42. sc11-app/slp-synthesis-aes-bottom13.cnf
43. sc11-app/sokoban-sequential-p145-microban-sequential.030-NOTKNOWN.cnf
44. sc11-app/traffic_b_unsat.cnf
45. sc11-app/traffic_kkb_unknown.cnf
46. sc11-app/UR-10-10p1.cnf
47. sc11-app/UTI-20-10t1.cnf
48. sc11-app/vmpc_25.renamed-as.sat05-1913.cnf
49. sc13-app/001.cnf
50. sc13-app/002.cnf
51. sc13-app/003.cnf
52. sc13-app/006.cnf
53. sc13-app/007.cnf
54. sc13-app/aes_24_4_keyfind_4.cnf
55. sc13-app/aes_32_3_keyfind_1.cnf
56. sc13-app/aes_32_3_keyfind_2.cnf
57. sc13-app/aes_64_1_keyfind_1.cnf
58. sc13-app/AProVE07-02.cnf
59. sc13-app/arcfour_initialPermutation_6_14.cnf
60. sc13-app/bivium-39-200-0s0-0x1b770901581bbb2863c83835583d7ce4e1fafd907076320542-34.cnf
61. sc13-app/bivium-39-200-0s0-0x28df9231b320bd56dfb68bfc7c3f0ca20dbae6b0eba535ad91-98.cnf
62. sc13-app/bivium-39-200-0s0-0x5fa955de2b4f64d00226837d226c955de4566ce95f660180d7-30.cnf
63. sc13-app/bivium-39-200-0s0-0xdcfb6ab71951500b8e460045bd45afee15c87e08b0072eb174-43.cnf
64. sc13-app/ctl_3082_415_unsat.cnf
65. sc13-app/ctl_4291_567_10_unsat.cnf
66. sc13-app/ctl_4291_567_11_unsat.cnf
67. sc13-app/ctl_4291_567_1_unsat.cnf
68. sc13-app/ctl_4291_567_1_unsat_pre.cnf
69. sc13-app/ctl_4291_567_7_unsat_pre.cnf

70. sc13-app/ctl_4291_567_8_unsat_pre.cnf
71. sc13-app/gss-17-s100.cnf
72. sc13-app/gss-18-s100.cnf
73. sc13-app/gss-19-s100.cnf
74. sc13-app/p01_lb_05.cnf
75. sc13-app/pb_200_03_lb_01.cnf
76. sc13-app/pb_200_03_lb_02.cnf
77. sc13-app/pb_200_05_lb_00.cnf
78. sc13-app/pb_200_10_lb_15.cnf
79. sc13-app/pb_400_03_lb_05.cnf
80. sc13-app/smtlib-qfbv-aigs-lfsr_004_127_112-tseitin.cnf
81. sc13-app/vmpc_29.cnf
82. sc13-app/vmpc_32.renamed-as.sat05-1919.cnf
83. sc13-app/vmpc_33.cnf
84. sc14-app/aes_24_4_keyfind_4.cnf
85. sc14-app/aes_32_3_keyfind_1.cnf
86. sc14-app/aes_64_1_keyfind_1.cnf
87. sc14-app/AProVE07-03.cnf
88. sc14-app/atco_enc1_opt1_15_240.cnf
89. sc14-app/atco_enc1_opt2_10_12.cnf
90. sc14-app/atco_enc1_opt2_10_14.cnf
91. sc14-app/atco_enc2_opt1_15_100.cnf
92. sc14-app/ctl_3791_556_unsat_pre.cnf
93. sc14-app/grieu-vmpc-31.cnf
94. sc14-app/gss-18-s100.cnf
95. sc14-app/k2fix_gr_rcs_w9.shuffled.cnf
96. sc14-app/MD5-27-4.cnf
97. sc14-app/q_query_3_l48_lambda.cnf
98. sc14-app/vmpc_29.cnf
99. sc14-app/vmpc_32.renamed-as.sat05-1919.cnf
100. sc14-app/vmpc_33.cnf