DISSERTATION

# DISCOVERING AND HARNESSING STRUCTURES IN SOLVING APPLICATION SATISFIABILITY INSTANCES

Submitted by

Wenxiang Chen

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Spring 2018

Doctoral Committee:

Advisor: L. Darrell Whitley

Bruce A. Draper
A. P. Wim Böhm
Edwin K. P. Chong

Copyright by Wenxiang Chen 2018

All Rights Reserved

# ABSTRACT

## DISCOVERING AND HARNESSING STRUCTURES IN SOLVING APPLICATION SATISFIABILITY INSTANCES

Boolean satisfiability (SAT) is the first problem proven to be NP-Complete. It has become a fundamental problem for computational complexity theory, and many real-world problems can be encoded as SAT instances. Two major search paradigms have been proposed for SAT solving: Systematic Search (SS) and Stochastic Local Search (SLS). SLS solvers have been shown to be very effective at uniform random instances; SLS solvers are consistently the top winning entries for random tracks at SAT competitions. However, SS solvers dominate hard combinatorial tracks and industrial tracks at SAT competitions, with SLS entries being at the very bottom of the ranking. In this work, we classify both hard combinatorial instances and industrial instances as *Application Instances*. As application instances are more interesting from a practical perspective, it is critical to analyze the structures in application instances as well as to improve SLS on application instances. We focus on two structural properties of SAT instances in this work: *variable interaction topology* and *subproblem constrainedness*.

*Decomposability* focuses on how well the variable interaction of an application instance can be decomposed. We first show that many application instances are indeed highly decomposable. The decomposability of a SAT instance has been extensively exploited with success by SS solvers. Meanwhile, SLS solvers direct the variables to flip using only the objective function, and are completely *oblivious* of the decomposability of application instances that is inherent to the original problem domain. We propose a new method to decompose variable interactions within SLS solvers, leveraging numerous visited local optima. Our empirical study suggests that the proposed method can vastly simplify SAT instances, which further results in decomposing the instances into thousands of connected components.

Furthermore, we demonstrate the *utility* of the decomposition, in improving SLS solvers. We propose a new framework called *PXSAT*, based on the recombination operator *Partition Crossover (PX)*. Given $q$ components, PX is able to find the best of $2^q$ possible candidate solutions in linear time. Empirical results on an extensive set of application instances show PXSAT can yield statistically significantly better results. We improve two of best local search solvers, AdaptG$^2$WSAT and Sparrow. PXSAT combined with AdaptG$^2$WSAT is also able to outperform CCLS, winners of several recent MAXSAT competitions.

The other structural property we study is *subproblem constrainedness*. We observe that, on some application SAT instance classes, the original problem can be partitioned into several subproblems, where each subproblems is highly constrained. While subproblem constrainedness has been exploited in SS solvers before, we propose to exploit it in SLS solvers using two alternative representations that can be obtained efficiently based on the canonical CNF representation. Our empirical results show that the new alternative representative enables a simple SLS solver to outperform several sophisticated and highly optimized SLS solvers on the SAT encoding of semiprime factoring problem.

# ACKNOWLEDGEMENTS

First and foremost I want to thank my advisor L. Darrell Whitley. It has been an honor to be his Ph.D. student. He has taught me, both consciously and unconsciously, how good research is done. The enthusiasm he has for his research was contagious and motivational for me, even during tough times in the Ph.D. pursuit.

I owe special thanks to my advisor, Adele E. Howe for her continuous support, inspiration, and guidance. She spent countless hours to help me make my pursuit of the Ph.D. a successful and fulfilling journey. Although she is no longer with us, she will be forever living in our minds.

I would like to thank my Ph.D. committee members Bruce A. Draper, A. P. Wim Böhm and Edwin K. P. Chong for their time and the valuable feedback.

Lastly, I would like to thank my family for all their love and encouragement. For my mom and dad who raised me with unconditional support even in difficult circumstances and trusted me in all my pursuits. And most of all, I would like to thank my girlfriend Qian for the countless laugh, the stimulating discussion on probability and statistics theory and the unending encouragement that enabled me to make it through the Ph.D. program.

TABLE OF CONTENTS


Abstract . . . ii
Acknowledgements . . . iv
List of Tables . . . vii
List of Figures . . . ix

Chapter 1 Introduction . . . 1
1.1 Satisfiability . . . 1
1.2 Motivation . . . 2
1.3 Contribution . . . 4

Chapter 2 Literature Review . . . 7
2.1 Backbone . . . 7
2.2 Backdoor . . . 10
2.2.1 Backdoor and SS Solvers . . . 10
2.2.2 Variable Dependency . . . 12
2.2.3 Summary . . . 16
2.3 Variable Interaction . . . 16
2.3.1 Variable Interaction Graph Visualization . . . 17
2.3.2 Complex Network Analysis . . . 18
2.3.3 Treewidth . . . 21
2.3.4 Summary . . . 25
2.4 SLS Search Space . . . 26
2.5 Identifying a Gap in Previous Works . . . 29

Chapter 3 Decomposing Variable Interaction Graphs with Pseudo Backbones . . . 31
3.1 Feasibility of Decomposing VIGs with Pseudo Backbones . . . 32
3.1.1 Identifying Application Instances with Potential Decomposability . . . 33
3.1.2 Computing Pseudo Backbone from Good Local Optima . . . 44
3.1.3 Improving Decomposition on SAT Instances . . . 48
3.2 Conclusion . . . 51

Chapter 4 Partition Crossover for Improving Stochastic Local Search SAT Solvers . . . 55
4.1 Variable Interaction and Tunneling . . . 57
4.1.1 Tunneling between Local Optima . . . 60
4.1.2 The Cost of PX compared to Local Search . . . 61
4.1.3 Equal Move for PX on Plateau . . . 63
4.2 PXSAT . . . 64
4.3 Empirical Results . . . 67
4.3.1 Setup . . . 67
4.3.2 Improving State-of-Art Local Search SAT Solvers . . . 68
4.3.3 Why and When PXSAT works? . . . 71

4.3.4 Competing with State-of-Art MAXSAT Solver . . . 75
4.4 Conclusions . . . 75

Chapter 5 Exploiting Subproblem Constrainedness with Alternative Representations . . 77
5.1 Related Work . . . 78
5.2 Exploit Subproblem Constrainedness using Conjunctive Minterm Canonical Form . . . 79
5.2.1 Sources for the Highly Constrained Subproblems . . . 80
5.2.2 Exploiting High Subproblem Constrainedness: From CNF to CMCF . . 82
5.2.3 Constraint Propagation over Minterms . . . 84
5.2.4 Local Search over CMCF . . . 86
5.2.5 Empirical Results on CMCF-LS . . . 88
5.3 Minterm Interaction Graph for Improving CMCF . . . 95
5.3.1 Increasing the Cardinality of Evaluation Function . . . 96
5.3.2 Reducing Running Time with Partial Updates . . . 103

Chapter 6 Conclusion and Future Work . . . 112
6.1 Future Work . . . 112
6.1.1 Partition Crossover for SLS SAT Solvers . . . 112
6.1.2 Local Search over Minterm Interaction Graphs . . . 113

Bibliography . . . 117

LIST OF TABLES

2.1 Summary of Structural Properties. "Inst-Spec" indicates whether the structural property is instance-specific. "NP-Hard" indicates whether acquiring the exact structural property is NP-Hard. "Heuristic" indicates whether polynomial-time heuristics is available. "SS Perm" (or "SLS Perm") indicates weather the structural property can be used to explain the performance of SS (or SLS) solvers. "Impr SS" (or "Impr SLS") indicates weather the structural property has been exploited to improve the performance of SS (or SLS) solvers. . . . . . . . . . 29

3.1 Detailed statistics on the 28 selected preprocessed application instances, sorted by *tw.pre.n*. "inst" indicates the instance name. "track" indicates the track from which the application instance is: "comb" means hard combinatorial track; "indu" means industrial track. "n.pre" indicates the number of variables. "m.pre" indicates the number of clauses. "tw.pre" indicates the treewidth. "tw.pre.n" indicates the normalized treewidth. 40

3.2 Quality of the best local optima found by AdaptG$^2$WSAT over 10 of 1000-second runs, on the 28 selected preprocessed application instances with small *tw.n.pre*. It is a minimization problem with 0 being a global optimum, where the evaluation of a candidate solution is the number of unsatisfied clauses. "inst" indicates the instance name. "n.pre" ("m.pre") indicates the number of variables (clauses). "eval.mean" ("eval.std") indicates the mean (standard deviation) evaluations of the best local optima over 10 runs. . . . . . . . . . 45

3.3 Statistics ("min.fixed", "mean.fixed", "std.fixed", and "max.fixed" columns) on numbers of fixed variable assignments in every pair of local optima, and the number of fixed variable assignments across all local optima ("all.fixed" column). . . . . . . . . . 47

3.4 Statistics ("min, "median and "max columns) on numbers of connected components after decomposition using pseudo backbones. . . . . . . . . . 50

4.1 An Example of MAX-3SAT Instance. . . . . . . . . . 57

4.2 Comparing PXSAT versions with the original local search solvers. "#No Diff (No PX)": number of instances where PX is never triggered. "#No Diff (PX Ties)": number of instances where PX is triggered and average solving time and average solution quality are tied between the original solver and its PXSAT version. "# Better (Worse) $\Delta sol$": number of instances where the PXSAT version has better (worse) average solution quality. "# Faster (Slower) $\Delta time$": number of instances where the PXSAT version has faster (slower) average solving time. . . . . . . . . . 69

4.3 Summary statistics on number of components ($q$) and PX Success Rate at finding improving moves for Sparrow-PX. . . . . . . . . . 73

4.4 Comparing AdaptG$^2$WSAT , Sparrow and their PXSAT versions with CCLS. "# Better (Worse) $\Delta sol$": number of instances where the PXSAT version has better (worse) average solution quality. "# Faster (Slower) $\Delta time$": number of instances where the PXSAT version has faster (slower) average solving time. . . . . . . . . . 75

5.1 Demonstration of Calculation of MinVote evaluation of candidate solution $(s_{1,2}, s_{2,2})$ for $\mathcal{F}'$ as shown in Figure 5.1. . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
5.2 Problem instance sizes as measured by number of variables, number of clauses for CNF, number of blocks for CMCF encodings, and average number of CNF clauses per Block. 89

LIST OF FIGURES


1.1 Ranking of solvers in Hard Combinatorial SAT track of SAT Competition 2014. http://satcompetition.org/edacc/sc14/experiment/21/ranking/ . . . . . . . . . . . . . . . . . . . 4
1.2 Ranking of solvers in Industrial SAT track of SAT competition 2014. http://satcompetition.org/edacc/sc14/experiment/18/ranking/ . . . . . . . . . . . . . . . . . . . 5

2.1 A uniform random 3SAT instance after three decisions in a systematic search [1]. . . . 19
2.2 A graph with eight vertices (left subfigure), and an optimal tree decomposition of it onto a tree with six nodes (right subfigure). $Treewidth = 3 - 1 = 2$. . . . . . . . . . . 22

3.1 The impact of preprocessing on the treewidths (computed by the Minimum Degree heuristic) of 128 satisfiable instances from industrial tracks of SAT Competition 2014. The diagonal line is the $y = x$. Each point represents a pair of treewidths of an original instance and the corresponding preprocessed instance. Points below (above) indicates the treewidth of the original instance is larger (smaller). . . . . . . . . . . . . . . . . . 35
3.2 The impact of preprocessing on the treewidths (computed by the Minimum Degree heuristic) of 128 satisfiable instances from hard combinatorial tracks of SAT Competition 2014. The diagonal line is the $y = x$. Each point represents a pair of treewidths of an original instance and the corresponding preprocessed instance. Points below (above) indicates the treewidth of the original instance is larger (smaller). . . . . . . . . . . . 37
3.3 Normalized Treewidths of 258 preprocessed application instances. Each bar counts the frequency of the normalized treewidths being within an interval of size 0.01. . . . . . . 39
3.4 Variable interaction graphs of preprocessed application instances atco_enc3_opt1_13_48 (top) and LABS_n088_goal008 (bottom). . . . . . . . . . . . . . . . . . . . . . . . . 41
3.5 Variable interaction graphs of preprocessed application instances SAT_instance_N=49 (top) and aaai10-ipc5 (bottom). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
3.6 Variable interaction graphs of preprocessed application instances: prime2209-98 (top) and AProVE09-06 (bottom). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
3.7 Decomposed Variable interaction graphs of representative application instances, atco_enc3_opt1_13_48 (top) and LABS_n088_goal008 (bottom), that yields the median number of connected components. . . . . . . . . . . . . . . . . . . . . . . . . 52
3.8 Decomposed Variable interaction graphs of representative application instances, SAT_instance_N=49 (top) and aaai10-ipc5 (bottom), that yields the median number of connected components. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
3.9 Decomposed Variable interaction graphs of representative application instance AProVE09-06 that yields the median number of connected components. . . . . . . . . 54

4.1 An illustration of the VIG. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
4.2 The Recombination Graph with three separable recombining components for the parent $P1 =$ 00000 00000 00000 000 and $P2 =$ 11100 01110 11101 101. . . . . . . . . . . . 59

4.3 This figure illustrates how PX is combined with local search. The dashed line tracks changes in $f(x)$. When a plateau is reached, a solution $P1$ is captured. After $\alpha n$ moves with no improving moves, another solution $P2$ is selected, and PX is used to recombine $P1$ and $P2$. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
4.4 Comparing PXSAT versions with the original local search solvers on instances where the average solution quality differences are statistically significant. . . . . . . . . . . . 70
4.5 Empirical $\overline{\Delta sol}$ versus theoretically approximated $\check{lr}$ for Sparrow. Left subfigure has a scale of [-50,50] on X-axis, while right subfigure has a scale of [50,650] on X-axis. Points with $\overline{\Delta sol} > 0$ ($\overline{\Delta sol} < 0$) are colored **blue** (**red**). . . . . . . . . . . . . . . . . . 72
4.6 The VIG (top) and the decomposed recombination graph (bottom) for SAT_instance_N=111. In this instance, tunneling will return the best of $2^{842}$ solutions. . . . . . . . . . . . . . 74

5.1 Satisfying minterms (the set of all partial solutions) to $M_1$ and $M_2$ of $\mathcal{F}'$ shown in Snippet 1, respectively. Each row represents a satisfying minterm to the block. A "1" under Boolean variable $b_i$ means the $b_i$ is set true in the partial solution, otherwise $b_i$ is set false; $s_{i,j}$ is the $j$th partial solution to $M_i$. . . . . . . . . . . . . . . . . . . . . . . . . 82
5.2 An example illustrating value constraint propagation. Each row represents a satisfying minterm to the block. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
5.3 An example illustrating equivalency constraint propagation. Each row represents a satisfying minterm to the block. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
5.4 Search space size on log-10 scale before and after CMCF-based constraint propagations on semiprime factoring problem instances. . . . . . . . . . . . . . . . . . . . . . . . . . 90
5.5 Runtime overhead for CMCF-based constraint propagations on semiprime factoring problem instances as a function of the number of variables in the original CNF version. The line is a linear least squared fit with adjusted R-squared of 0.8241. . . . . . . . . . 91
5.6 Success Rate on 50 Semiprime Factoring Problem Instances. The two trend curves are constructed using local polynomial regression fitting. . . . . . . . . . . . . . . . . . . . 92
5.7 Success rate for the five parity learning instances. . . . . . . . . . . . . . . . . . . . . . 93
5.8 Number of Evaluations spent by CMCF-LS and WalkSAT on Semiprime Instances. Boxplot shows the distribution over 50 runs. Lines connect the medians. . . . . . . . . 94
5.9 Number of Evaluations for CMCF-LS and WalkSAT on parity instances. Boxplot shows the distribution over 50 runs. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
5.10 CPU Time spent by CMCF-LS and WalkSAT on Semiprime Instances. Boxplot shows the distribution over 50 runs. Lines connect the medians. . . . . . . . . . . . . . . . . . 96
5.11 CPU Time spent by CMCF-LS and WalkSAT on parity instances. Boxplot shows the distribution over 50 runs. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
5.12 An example illustrating arc consistency propagation. Each row represents a satisfying minterm to the block. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 100
5.13 Success Rate on 50 Semiprime Factoring Problem Instances. The two trend curves are constructed using local polynomial regression fitting. . . . . . . . . . . . . . . . . . . . . 102
5.14 Number of Evaluations spent by CMCF-LS and Gforce on Semiprime Instances. Boxplot shows the distribution over 50 runs. Lines connect the medians. . . . . . . . . . . 102
5.15 Number of Evaluations for CMCF-LS and Gforce on parity instances. Boxplot shows the distribution over 50 runs. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103

5.16 Relationship between semiprime to factor and the number of Boolean variables needed to encode the semiprime factoring problem in the CNF representation. . . . . . . . . . . 108
5.17 Success rate of on 27 semiprime factoring instances. . . . . . . . . . . . . . . . . . . . . . 110
5.18 CPU time in seconds of Gforce and best performing SLS solvers on 27 semiprime factoring instances. Boxplot shows the distribution over 10 of one hour run. . . . . . . 111

# Chapter 1

# Introduction

Given a Boolean Formula $\mathcal{F}$, Boolean Satisfiability (SAT) is the problem of deciding whether there exists an assignment $\mathcal{A}$ to the Boolean variables such that $\mathcal{F}$ is true. SAT is the first problem proven $\mathsf{NP}$-Complete [2]. SAT has become a fundamental problem for computational complexity theory. Any $\mathsf{NP}$-Complete problem can be reduced in polynomial time to SAT and solved by a SAT solver. In fact, SAT has also been used as the “gateway” problem for new problems (such as the independent set problem) to enter the $\mathsf{NP}$-Complete class by constructing a polynomial time reduction from the new problem to SAT [3, 4]. Besides its theoretical importance, SAT also finds many practical applications such as bounded model checking [5] and verification in hardware [6] and software [7]. Erdős Discrepancy Conjecture, a longstanding mathematical conjecture proposed by the famous mathematician Paul Erdős in 1930s, has recently been attacked successfully using a SAT solver [8, 9].

## 1.1 Satisfiability

SAT problem instances are usually defined in Conjunctive Normal Form (CNF) : a conjunction of clauses $f = \bigwedge_{c_i \in \mathbb{C}} c_i$, where each clause $c_i$ is a disjunction of literals $c_i = \bigvee_{l_j \in \mathbb{L}_i} l_j$ and each literal is either a Boolean variable $b$ or its negation $\bar{b}$. *MAX-SAT* is an optimization version of SAT problem. The goal of MAX-SAT is to find an variable assignment such that the number of satisfied clauses is maximized. A more restricted form of MAX-SAT, called *MAX-kSAT*, requires the number of literals in every clause is exactly $k$.

DIMACS format is the standard file format used to succinctly represent CNF instances [10]. With DIMACS format, each line represents a clause with a “0” as end-of-line delimiter. A clause is defined by listing the index of each positive literal, and the negative index of each negative literal.

This way, conjunction operator $\bigwedge$ and disjunction operator $\bigvee$ are implicitly encoded with the space between literals in a line and the "0" between lines.

Listing 1.1 shows a simple CNF instance represented in DIMACS format. First line starting with the character "c" is comment and is ignored by parsers. Line 2 starting with the character "p" is a line shows that the problem has 3 variables and 2 clauses. Line 3 and Line 4 is the one actual clause, representing the CNF instance $(b_1 \vee \overline{b_3}) \wedge (b_2 \vee b_3 \vee \overline{b_1})$.

**Listing 1.1:** A Simple CNF Instance in DIMACS Format.

```
c  simple_v3_c2.cnf
p cnf 3 2
1 -3 0
2 3 -1 0
```

## 1.2 Motivation

The two major search paradigms for SAT solving are 1) Systematic Search (SS) such as Davis–Putnam–Logemann–Loveland (DPLL) [11], zChaff [12] and MiniSat [13], and 2) Stochastic Local Search (SLS) such as GSAT [14], WalkSAT [15], AdaptG$^2$WSAT [16] in UBCSAT collection [17], and the more recent Configuration Checking with Aspiration (CCA) [18]. The NP-completeness of SAT indicates that there is no known polynomial time algorithm; SAT solving can take exponential time in the number of variables in the worst case.

The international SAT competition [19] is an annually held competition to keep up the driving force in improving SAT solvers and to present the latest technical advancements to a broader audience. The top SLS solvers in recent SAT competitions can reliably solve uniform random 3SAT instances with 1 million variables and several million of clauses around the phase transition. Literals for each clause are sampled uniformly randomly without replacement from all $2n$ literals . These uniform random instances are in expectation the hardest around the phase transition region [20–22]. There are two other tracks in the SAT competitions: the Hard Combinatorial Track and the Industrial Track. The *Hard Combinatorial Track* consists of instances encoding combinatorial problems that

are known to be difficult, such as the factoring problem [23] and the clique coloring [24]; The *Industrial Track* consists of instances encoding industrial applications, such as cryptography [25] and hardware verification [6]. The same state-of-art SLS solvers display drastically poorer performance on the hard combinatorial track and the industrial track of SAT competitions. SS solvers instead dominate hard combinatorial tracks and industrial tracks at SAT competitions, with all SLS entries being at the very bottom of the ranking in terms of performance.

Uniform random instances are interesting mostly from a theoretical point of view. The principled distribution used in generating uniform random instances make them a good platform for theoretical analysis [26]. Hard combinatorial instances and industrial instances on the other hand provides a better testbed for evaluating the feasibility of using SAT approach under industrial settings [27, 28]. In this work, we classify both hard combinatorial instances and industrial instances as *Application Instances*. Application instances often have structure to them, which can be a result of the loosely coupled components originated from the source problem domain [29, 30]; and/or the procedure involved in translating a problem from the original domain into a SAT instance [31, 32]. It is critical to identify the problem structures in applications as well as to improve SLS on application instances.

Improving stochastic local search on structured problems by efficiently handling variable dependencies has been considered one of the fundamental challenges in propositional reasoning and search since 1997 [33, 34]. After almost 20 years, according to the recent SAT competitions [19], the current SLS solvers still struggle on application instances, and their performance is still dominated by the SS solvers by a substantial margin. For instance (see Figure 1.1), the best SLS solver in the Hard Combinatorial SAT track of SAT competition 2014 "`CPSparrow sc14`" solves 56 of out 150 instances; The worst SS solver in the same track "`SatUZK`" solve 71 instances, while the best solver in track "`SparrowToRiss 2014`" solves 107 instances. Interestingly, the worst 6 qualified entries (excluding 4 disqualified ones at the bottom of Figure 1.1) in the track are all SLS solvers.

The gap between the SLS solvers and the SS solvers become even more pronounced in the industrial track. In the Application SAT track of SAT competition 2014, there are only 3 standalone SLS

| # | Solver | # of successful runs | % of all runs | % of VBS runs | Cumulated cost | Average cost |
|---|---|---|---|---|---|---|
| 26 | BFS-Glucose_mem_16_70 1.0cm | 78 | 52.0 % | 62.9 % | 396534.634 | 2643.564 |
| 27 | ntusatbreak 1.0.0 | 77 | 51.33 % | 62.1 % | 405982.083 | 2706.547 |
| 28 | ROKK 1.0.1 | 77 | 51.33 % | 62.1 % | 427500.747 | 2850.005 |
| 29 | ntusat 1.0 | 75 | 50.0 % | 60.48 % | 403593.256 | 2690.622 |
| 30 | BFS-Glucose_mem_32_70 1.0cm | 75 | 50.0 % | 60.48 % | 409066.039 | 2727.107 |
| 31 | BFS-Glucose_mem_8_85 1.0cm | 75 | 50.0 % | 60.48 % | 410075.383 | 2733.836 |
| 32 | BFS-Glucose 1.0c | 75 | 50.0 % | 60.48 % | 410153.168 | 2734.354 |
| 33 | MSP 1.0 | 75 | 50.0 % | 60.48 % | 434999.045 | 2899.994 |
| 34 | SatUZK 55 | 71 | 47.33 % | 57.26 % | 442246.665 | 2948.311 |
| 35 | CPSparrow sc14 | 56 | 37.33 % | 45.16 % | 476956.822 | 3179.712 |
| 36 | YalSAT 03l | 48 | 32.0 % | 38.71 % | 523870.166 | 3492.468 |
| 37 | sattime2014r 2014r | 47 | 31.33 % | 37.9 % | 528614.389 | 3524.096 |
| 38 | probSAT sc14 | 34 | 22.67 % | 27.42 % | 601841.925 | 4012.28 |
| 39 | BalancedZ SAT Competition 2014 64-bit V2014.05.07 | 34 | 22.67 % | 27.42 % | 603015.54 | 4020.104 |
| 40 | CCA2014 2.0 | 32 | 21.33 % | 25.81 % | 600011.623 | 4000.077 |
| 41 | miniLoCeG(2SAT)+glucose (disqualified) 1.0 | 2 | 1.33 % | 1.61 % | 195145.189 | 4759.639 |
| 42 | miniLoCeG(UNIT)+glucose (disqualified) 1.0 | 2 | 1.33 % | 1.61 % | 195145.725 | 4759.652 |
| 43 | provoSATeur+glucose (disqualified) 1.0 | 2 | 1.33 % | 1.61 % | 195149.073 | 4759.733 |
| 44 | glueminisat | 0 | 0.0 % | 0.0 % | 750000.0 | 5000.0 |

Showing 1 to 44 of 44 entries

**Figure 1.1:** Ranking of solvers in Hard Combinatorial SAT track of SAT Competition 2014. http://satcompetition.org/edacc/sc14/experiment/21/ranking/

solvers entering the track (see Figure 1.2). The best SLS solver is again "`CPSparrow sc14`" that solves 29 instances out of 150 instances, tied with "`sattime2014r`"; The worst SS solver "`BFS-Glucose_mem_32_70 1.0cm`" solves 79 instances and the best SS solver "`minisat_blbd 1.13`" solves 110 instances. Improving SLS on application instances remains a difficult challenge.

## 1.3 Contribution

We aim to address the following research questions:

1. What characteristics differentiate application instances from uniform random instances?

2. How can we leverage problem structure to improve SLS solvers on application instances?

In order to address the research questions, we focus on analyzing two structural properties:

1. Variable interaction graph [1, 35],

2. Subproblem constrainedness [36],

| # | Solver | # of successful runs | % of all runs | % of VBS runs | Cumulated cost | Average cost |
|---|---|---|---|---|---|---|
| 22 | Lingeling ayv | 92 | 61.33 % | 71.88 % | 361787.849 | 2411.919 |
| 23 | SatUZK 55 | 91 | 60.67 % | 71.09 % | 359669.332 | 2397.796 |
| 24 | Lingeling (no agile) ayv | 91 | 60.67 % | 71.09 % | 366792.48 | 2445.283 |
| 25 | cryptominisat-4.1-st 4.1-st | 90 | 60.0 % | 70.31 % | 364623.047 | 2430.82 |
| 26 | cryptominisat-v41-tuned-st v41-tuned-st | 88 | 58.67 % | 68.75 % | 359891.389 | 2399.276 |
| 27 | glue_lgl_split 1.0 | 87 | 58.0 % | 67.97 % | 381621.159 | 2544.141 |
| 28 | BFS-Glucose_mem_16_70 1.0cm | 84 | 56.0 % | 65.63 % | 433168.936 | 2887.793 |
| 29 | ntusatbreak 1.0.0 | 83 | 55.33 % | 64.84 % | 427008.657 | 2846.724 |
| 30 | BFS-Glucose_mem_8_85 1.0cm | 83 | 55.33 % | 64.84 % | 442362.206 | 2949.081 |
| 31 | BFS-Glucose 1.0c | 83 | 55.33 % | 64.84 % | 442390.946 | 2949.273 |
| 32 | ntusat 1.0 | 82 | 54.67 % | 64.06 % | 430015.591 | 2866.771 |
| 33 | BFS-Glucose_mem_32_70 1.0cm | 79 | 52.67 % | 61.72 % | 441785.117 | 2945.234 |
| 34 | CPSparrow sc14 | 29 | 19.33 % | 22.66 % | 622081.956 | 4147.213 |
| 35 | sattime2014r 2014r | 29 | 19.33 % | 22.66 % | 623545.039 | 4156.967 |
| 36 | YalSAT 03l | 25 | 16.67 % | 19.53 % | 635718.946 | 4238.126 |
| 37 | miniLoCeG(UNIT)+glucose 1.0 | 0 | 0.0 % | 0.0 % | 0.0 | 0.0 |
| 38 | miniLoCeG(2SAT)+glucose 1.0 | 0 | 0.0 % | 0.0 % | 0.0 | 0.0 |
| 39 | provoSATeur+glucose 1.0 | 0 | 0.0 % | 0.0 % | 210000.0 | 5000.0 |
| 40 | glueminisat | 0 | 0.0 % | 0.0 % | 750000.0 | 5000.0 |

**Figure 1.2:** Ranking of solvers in Industrial SAT track of SAT competition 2014. http://satcompetition.org/edacc/sc14/experiment/18/ranking/

First, we propose to visualize the Variable Interaction Graphs [1] (VIG for short; the visualization is also referred to as Variable Incidence Graph [35]) of the application instances. Treewidth is a metric that indicates the decomposability of VIG. SS solvers have long exploited the decomposability of application instances with success. We conjecture that SLS solvers can be improved by exploiting the decomposability of application instances, and propose the first step toward exploiting decomposability with SLS solvers using pseudo backbones [37]. We then propose two SAT-specific optimizations that lead to provably better decomposition than on general pseudo Boolean optimization problems. Our empirical study suggests that pseudo backbones can vastly simplify SAT instances, which further results in decomposing the instances into thousands of connected components. This finding serves as a key stepping stone for applying the powerful recombination operator, partition crossover, to SAT domain.

We further propose a generic framework, *PXSAT*, based on PX that demonstrate the utility of the decomposition. It applies Partition Crossover (PX) to local optima that are deemed difficult to improve by local search. PX is a recent powerful recombination operator that decomposes a

given instance into independent components, and is guaranteed to find the best solution among an exponential number of candidate solutions in linear time. It thus offers new opportunities to *jump to new local optima*. Empirical results on an extensive set of application instances show that the proposed framework substantially improves two of best local search solvers (AdaptG$^2$WSAT and Sparrow) on many application instances and almost never worsens the performance. PXSAT combined with AdaptG$^2$WSAT is also able to outperform CCLS, winner of several recent MAXSAT competitions. The new framework is orthogonal to any specific local search, therefore it can be combined with any local search.

Finally, we observe that Conjunctive Normal Form (CNF) SAT encodings of application instances often have a set of consecutive clauses defined over a small number of Boolean variables, which leads to high *subproblem constrainedness* [36]. To exploit the high subproblem constrainedness in the context of SLS solvers, we propose a transformation of CNF to an alternative representation, *Conjunctive Minterm Canonical Form (CMCF)* [38]. We show empirically that a simple SLS solver based on CMCF (CMCF-LS) can consistently achieve a higher success rate using fewer evaluations than the SLS solver WalkSAT [15] on two representative classes of application SAT problem. We further improve the CMCF representation by introducing Minterm Interaction Graph (MIG), which effectively increases the cardinality of the objective function. With MIG representation, a straightforward SLS solver inspired by WalkSAT called *Gforce* solves semiprime instances thousands of times faster than the highly optimized WalkSAT, scales better than sophisticated SLS solvers SAPS [39] and AdaptG$^2$WSAT [16], and compete well against the best SLS solver Sparrow [40] in recent competitions, in terms of the raw CPU time.

# Chapter 2

# Literature Review

This chapter presents a review on the structural properties of SAT instances investigated by previous work. We focus on revisiting the previous efforts to address the two major research questions that we proposed to study (see Section 1.3). A gap among the previous studies is identified at the end of the chapter.

While structures within uniform random instances (in terms of clauses-to-variables ratio) has been extensively studied both theoretically [22, 41] and experimentally [20, 42], we focus our literature review on these studies directly relate to the two research questions that we aim to address. Specially, the literature review covers the studies on 1) the differences in structures between uniform random instances and application instances, 2) how the structural differences help to explain the performance difference between SS solvers and SLS solvers, and 3) how to leverage the structures to improve SLS on applications instances.

For the purpose of understanding the performance of SAT solvers on diverse problem instances, numerous researchers have proposed and studied various structural properties from different perspectives. In this section we present a *taxonomy* over these seemingly scattered structural properties. We will review the structural properties from two perspectives: 1) the cost of discovering the structural property and 2) the utility of the structural properties.

## 2.1 Backbone

Monasson *et al*. introduce the concept *backbone* from statistical physics to describe the set of variables that have fixed values in all optimal solutions [22]. Computing a backbone intuitively means obtaining all global optima and finding the shared variable assignment across all global optima. This is of course expensive since finding all global optima is at least as hard as finding one single global optimum, which is NP-Hard. Kilby *et al*. prove that even approximating the backbone within a constant factor is NP-hard [43].

Various heuristics are available for computing an approximate backbone. Dubois and Dequen propose to compute a relaxed backbone associated with a subset of clauses [44]. By limiting the consideration to a subset of clauses, the backbone computation becomes tractable. One advantage of this approach is that the true backbone can only be a subset of relaxed backbone. Hsu *et al.* apply probabilistic message-passing algorithms to estimate the backbone variables [45,46], which result in higher estimation accuracy than Dubois and Dequen's method. The accuracy for approximating the assignment of backbone by Hsu's method can be as low as 70% even with their best approximation algorithm. While 70% might seem decent, the real problem is that any wrong assignment on a backbone variable rules out all global optima, and it is unknown apriori which of the 70% variables are assigned correctly. This empirical result conforms with the theoretical founding by Kilby *et al.* that backbones are hard even to approximate. Zhang, Rangan and Looks [47,48] instead propose to approximate global optima using local optima. The local optima are then be used to extract an approximate backbone. However, the accuracy of the pseudo backbone has not been studied in [47,48].

Besides heuristics, exact methods are available for backbone computation that require multiple calls to a SAT solver [49]. Even though optimization can be made to reduce the number of calls and share useful information between calls to improve efficiency, the exact method is not suitable for improving SAT solving and is mostly of interest for applications where the exact backbone is required such as product configuration [50].

Backbone information can be useful in two ways. First, the practical hardness of a SAT instance can be explained using the backbone size [22]. As setting any backbone variable wrong can exclude all global optimum, a larger backbone size suggests that global optima can be harder to find [51]. On instances with a large backbone, SS solvers have many opportunities to make mistakes and to waste time searching subspaces without any global optimum before correcting the wrong assignments; SLS solvers can also struggle to find global optima because they are in expectation either scarce or clustered, due to the large backbone size [52,53]. Kilby *et al.* [43] reports that backbone size is only weakly correlated (correlation coefficients between 0.16 and 0.48) with the solving time of larger

uniform random instances (with number of variables ranging from 100 to 225). On smaller uniform random instances, the correlation becomes strongly negative (correlation coefficients between -0.46 and -0.88). Instance hardness in [43] is measured by the log of the number of search node by the SS solver "Satz" [54]. Conversely, the correlation for SLS solvers appears more significant. Based on $10^4$ satisfiable instances sampled from phase transition region with 200 variables, Parkes [52] shows that larger backbone size leads to drastic increase in median search time of the SLS solver WalkSAT [15].

Second, backbone information can also be exploited to boost both the performance of both SS solvers and SLS solvers. Dubois and Dequen improve the performance of a SS solver by guiding variable selection with relaxed backbone on uniform random k-SAT instances [44, 55]. Their approach aims to minimize the search tree by selecting the variable that are most constrained and therefore are more likely to be backbone variables. By assuming a uniform distribution for constrainedness, Dubois and Dequen's approach is specialized for uniform random instances. Indeed, their solver is the winning SS solver of SAT'03, SAT'04 and SAT'05 competitions in the category Random Benchmarks. How well can Dubois and Dequen's approach generalize to application instances is unknown. With improved accuracy of backbone estimation over Dubois and Dequen's approach, VARSAT by Hsu *et al.* manages to achieve up to $10\times$ speedup over MiniSAT [13] also on uniform random instances by exploiting probabilistically estimated backbones [45, 46]. Incorporating estimated backbone information appears to find little success in improving SS solvers on application instances, whereas substantial improvement can be observed on application instances for SLS solvers. Zhang, Rangan and Looks [47, 48] guide bit flips in the SLS solver WalkSAT [15] with pseudo backbone estimated from local optima. The backbone guided SLS solver yields higher success rates in finding a satisfiable solution on uniform random instances and better solution quality (20% better on average) on application instances from the MAXSAT standpoint.

## 2.2 Backdoor

### 2.2.1 Backdoor and SS Solvers

Modern SS solvers can solve application instances with up to millions of variables, which is far beyond the limit suggested by the exponential worst-case complexity. For the purpose of advancing the understanding of the typical-case complexity of modern SS solvers on application instances, Williams, Gomes, and Selman propose the *backdoor* concept [56]. By definition, a *weak backdoor* is the set of variables such that when assigned correctly, a sub-solver can solve the remaining problem in polynomial time; a *strong backdoor* is the set of variables such that when assigned (in any arbitrary way), the remaining problem becomes solvable (proven either SAT or UNSAT) in polynomial time. A trial backbone is the set of all variables, as setting all variables correctly will automatically solve the remaining empty problem. An instance with a small backdoor suggests that there exists shortcuts to attack the entire problem.

Finding the smallest backbone in general is NP-Hard [43, 56–58]. Note that backdoor is defined with respect to a subsolver, which in turn can be defined algorithmically or syntactically [59]. Algorithmically-defined subsolvers are polynomial-time techniques of SAT solvers like *unit propagation*, which propagates the assignments to single literals to simplify the Boolean formula. Syntactically-defined subsolvers are known tractable classes, such as 2-SAT (each clause contains at most two literals) and HORN (each clause contains at most one positive literal). If the backbone size is known to be bounded, finding a syntactically-defined strong backdoor into tractable classes 2-SAT and HORN *is tractable*, whereas finding a weak backdoor is not [60]. Finding a algorithmically-defined weak backdoor typically involves finding a solution using a complete solver, saving the decision variables (rather than the variables implied by propagation), and finally simplifying the set of decision variables to obtain a smaller weak backdoor (SATZWEAK algorithm in [43]). Finding a algorithmically-defined strong backdoor, in comparison, simply tests every combination of literals up to a fixed cardinality (STRONGBACKDOOR algorithm in [43]). While it might look promising that finding strong backdoors to 2-SAT and HORN is tractable, the found syntactically-defined

strong backdoors are generally larger than algorithmically-defined strong backdoors, which limits the utility of the found strong backdoors [61].

In fact, without the concept of backdoor in mind, SS solvers such as Satz-Rand [54, 62] already implicitly search for a backdoor by prioritizing those variables that cause a large number of unit-propagations and therefore simplifies the remaining problem to a tractable subproblem. Satz-Rand is remarkably efficient at detecting small strong backdoors. Satz-Rand finds strong backdoors that are in size less than 1% of the total number of variables on diverse application instances [61]. It is important to note that the small strong backdoors detected are merely upper bounds on the smallest strong backdoors. Li and Van Beek [59] propose an exact algorithm to find weak backdoor. They found that application instances up to 1000 variables have smallest weak backdoors of size up to 3. This suggests that many applications instances do have small backdoors.

On the other hand, there exist instances that *do not* have small backdoors, such as uniform random instances or instances based on cryptographic protocols [63]. Such instances appear to be inherently hard for SS solvers. For uniform random 3SAT problems, the backdoor appears to be a constant fraction (roughly 30% to 60% depending on the clause-variable ratio) of the total number of variables. This may explain why the current SS solvers have not made significant progress on hard uniform random instances [64].

Even though the original intention of the backdoor concept is to explain the performance of SS solvers on application instances, a limited number of statistical studies has been conducted on the correlation between running time of SS solvers and the backdoor size of applications instances. Ruan, Kautz, and Horvitz [65] report that there is no significant correlation between weak backdoor size and instance hardness measured by the log of the median running time of SS solver Satz-Rand. The only other statistical study that we are aware of is conducted by Kilby, Slaney, Thiébaux, and Walsh [43]. They suggest that instance hardness appears to be correlated with the strong backdoor size, and does not appear to be correlated with the weak backdoor size. Nonetheless, even the strong backdoor size, which shows the best correlation, only shows correlation coefficients ranging from 0.37 to 0.78 on uniform random instances. No statistical correlation study were carried out by Kilby,

Slaney, Thiébaux, and Walsh for application instances due to the prohibitively high cost in detecting a strong backdoor.

In addition to its utility in explaining the running time of SS solvers, Williams, Gomes, and Selman [63] also establish the connection between heavy trail behavior [66] in SAT solving time and backdoors. Having small backdoors in application instances explains how a SS solver can get "lucky" on certain runs, where backdoor variables are identified early on in the search and set the right way.

Finally, backdoors can be utilized to improve both the theoretical complexity and practical performance of SS solvers. Given a problem with a small backdoor (i.e., backdoor size is $O(log(n))$, where $n$ is the number of variables), Williams, Gomes and Selman [56] prove that a variable selection heuristic with restart strategy can solve the problem in polynomial time. The polynomial bound improves over the $2^n$ ($n$ is the number of Boolean variables) worst-case complexity by canonical SS solvers. Kottler, Kaufmann, and Sinz [67] exploit backdoors for a NP-Hard subclass to obtain a better upper bound, $O(1.427^n * p(n))$ ($p(n)$ is a polynomial in $n$), over the best upper bound for deterministic algorithms $O(1.4424^n)$. Paris, Ostrowski, Siegel, and Sais demonstrate that the practical performance of the SS solver Zchaff [12] can be improved by branching only on the strong backdoor variables [58]. However, the (presumably large) overhead for detecting strong backdoors is not disclosed by the authors.

Backbones remain an active research area since the emergence of the concept in 2003. Recent efforts refine the backbone concept either to improve its instance hardness prediction power [68] or to find a smaller backbone involving less variables [69, 70].

### 2.2.2 Variable Dependency

Backdoors shed light on why SS solvers performs so well on many application instances, but not on uniform random instances. Interestingly, while backdoors can be exploited by modern SS solvers, the existence of a special kind of strong backdoor, caused by variable dependency, has been shown to impede SLS solvers [71].

Encoding problems from other domains into SAT often introduces *dependent* variables [72], whose values are defined by a Boolean function of other variables. These dependent variables (also called "auxiliary variables") are usually required by the well-known Tseitin encoding [73] to achieve linear size conversion of propositional logic formulas to Conjunctive Normal Forms (CNFs). In contrast, *independent* variables are those whose values cannot be determined as a simple Boolean function of other variables. Independent variables are often the variables native to the original problem before being converted to CNF. For example, the independent variables in the application instance encoding a planning problem represent the various operators applicable in a given state of the world, whereas the dependent variables encode the consequences of selecting a particular operator [56]. Independent variables are a kind of strong backdoor in the sense that once they are assigned, the values of the dependent variables can be determined accordingly and the entire formula can be evaluated in polynomial time.

SS solvers can handle variable dependencies effectively by propagating independent variables to dependent variables via unit propagation. Branching on dependent variables can be avoided by employing a branching heuristic that places the independent variables before dependent ones [74]. On the contrary, by its iterative nature, an SLS solver takes longer to propagate dependencies. Empirical results suggest that $O(n^2)$ ($n$ is the number of variables) steps are required for the values of the dependent variables to become aligned with independent variables [75]. Developing local search techniques that can effectively handle variable dependencies has been considered to be a fundamental challenge in propositional reasoning and search [33] [76].

Researchers have studied different ways of extracting various variable dependencies defined as logic gates such as AND gates and XOR gates, to reduce search cost on dependent variables. Note that there is a trade-off between two confounding factors. One factor is the types of target Boolean functions to extract, which reduces the search space exponentially for SLS solvers; the other factor is the time overhead required to extract them, which adds up to the overall search time. Indeed, Lang and Marquis [77] show that the decision problem, whether $y$ is a dependent variable with respect to Boolean function $f$ and independent variable $x_1, \ldots, x_i$ from a CNF formula (i.e.,

$y \stackrel{?}{\equiv} f(x_1, ..., x_i)$) is coNP-Complete. coNP class is similar to NP class in that there is no known polynomial time algorithm, with the difference being that a "Yes" answer in NP can be verified in polynomial time, whereas a "No" answer in coNP can be verified in polynomial time. Taking the Subset Sum problem as an example, the NP problem asks if there is a non-empty subset that sums to zero; verifying a "Yes" solution can be done in polynomial time. The coNP variant asks whether every non-empty subset sums to a non-zero number; verifying a "No" solution (counterexample) can be done in polynomial time. In case of the decision problem $y \stackrel{?}{\equiv} f(x_1, ..., x_i)$, a polynomial-time verifiable counterexample is a single assignment on $(x_1, ..., x_i)$ that corresponds to distinct values of $y$.

Extracting variable dependencies allows SLS solvers to focus search on independent variables and to align independent variables and their dependent variables in an explicit manner. Kautz, McAllester and Selman propose DAGSAT [72], which represents Boolean formulas as Directed Acyclic Graphs (DAGs) to encode the dependencies in a hierarchy. They define *dependent variables* as the outputs of "AND" gates and "OR" gates. They use a heuristic to find AND gates and OR gates in the CNF, and define the sink node (i.e., root variable) of the DAG to be the conjunction of all the clauses which are not parts of the heuristically recovered logic gates. The "heuristic" shall extract some of the logic gates within a reasonable amount of time. However, the details regarding the heuristic is undisclosed and there is no report on the theoretical nor empirical overhead of extracting variable dependencies. Empirical evaluation on DAGSAT show significant improvements (over $20\times$ speedup in solving time) over the SLS solver WalkSAT [15] and the SS solver NTAB [78] on instances with a large portion (up to 95%) of dependent variables. On instances that do not contain many dependent variables, however, DAGSAT is instead $680\times$ slower than WalkSAT.

Pham, Thornton and Satter [79][1] [80] extend DAGSAT [72] by also extracting "XNOR" gates and "XOR" gates in the processing phase, and develop a dependency lattice similar to the DAG in DAGSAT. By focusing search on the assignment of the independent variables, Pham *et al.* 's SLS solver improves upon the base SLS solver AdaptNovelty+ [81] on several classes of application

[1]The paper won best paper award at IJCAI 2007.

instances with large portion of dependent variables (72% to 99% of all variables). With only a fraction of a second runtime overhead in dependency extraction, Pham *et al.* 's SLS solver can solve instances up to $1400\times$ faster. Compared with improvement achieved by DAGSAT over the base solver WalkSAT (about 20 fold difference), considering more types of gates appear to boost the performance over the base solver more. This is also intuitive since extracting more dependencies result in less independent variables in expectation.

Explicitly extracting variable dependencies can also improve SS solvers in a significant way. Ostrowski *et al.* extend the dependency extraction in DAGSAT by also considering "XNOR" gates (also known as equivalence gates) [82]. To reduce the number of required syntactical test on clauses to extract logical gates, they construct a partial graph of clauses for detecting gates, since forming a connected subgraph is a necessary condition for clauses to belong to a same gate. The empirical study shows that it only takes 4.50 seconds on the largest tested instance with 5259 variables and 55424 clauses. The method successfully detects the exact 32 independent variables of parity learning problem that are native to the original problem domain [83], reducing the search space from $2^{3176}$ to $2^{32}$. Ostrowski *et al.* propose LSAT, which runs a SS solver on the remaining clauses and checks that the current assignment does not contradict the extracted dependencies. Paying the small upfront cost of dependency extraction turns into large improvement over state-of-the-art SS solvers. LSAT solves some instances in less than 1 second, whereas other SS solvers take more than 16 minutes. However, the empirical evaluation is somewhat unconvincing because two benchmark sets, one used by evaluating the dependency extraction preprocessor and the other used by evaluating LSAT as whole, only shares a small overlap. Ostrowski *et al.* 's dependency extraction techniques are later employed by the well-known SatELite [84], which in turn is the built-in preprocessor of the highly-influential SS solver MiniSAT [13].

Grégoire *et al.* [85] later discover that forming a connected subgraph is not truly a necessary condition for clauses belong to a same gate (see Example 2 in [85]). Employing Boolean Constraint Propagation enables extracting more gates. In additional, Grégoire *et al.* explore a heuristic to break cycles in variable dependencies. They further show that the improved dependency extraction

technique run in polynomial time. Empirically, the improved technique generally detects more gates. However, there is no report on the additional dependency detection cost induced by the more sophisticated technique, and how the increased number of detected gates translates into SAT solving performance is also unknown.

### 2.2.3 Summary

Independent variables typically appears less expensive to detect than backdoor variables. Independent variables can be detected locally only with respect to a (small) subset of clauses, such as the clauses forming a connected partial graph. On the contrary, backdoor detection by definition requires taking the entire instance into consideration. Williams, Gomes, and Selman report that backdoors are smaller than the set of independent variables [63]. For example, in the logistics planning domain, the set of independent variables is given by the number of operators applicable at each time step. This set of total variables will generally be much larger than the minimal backdoor set, which has only 12 variables. Nonetheless, detecting and respecting variable dependencies seems to be a reasonable first step for SLS solvers to catch up with SS solvers on application instances. Previous work only considers a very limited set of dependencies (at most four logic gates) and a synthesis of existing empirical studies suggest that taking more types of dependencies into account may further boost the performance of SLS solvers on application instances.

## 2.3 Variable Interaction

Although backbones and backdoors are somewhat correlated with instance hardness, they are computationally prohibitive to acquire and offers no apriori criteria to gauge the computational difficulty of a given instance. Variable interaction is another key property influencing the difficulty of SAT problem instances, and can be retrieved in polynomial time. Two variables *depend* if and only if they co-occur in some clause. In fact, given a SAT instance that only involves binary clauses (with just pairwise interactions), its satisfiability can be determined in polynomial time [86]. Moreover, experiments with uniform random instances from the $(2+p)$-model (problems with a fraction of

$p$ ternary clauses and $(1-p)$ binary clauses) indicate that uniform random instances with up to 40 percent of 3-clauses ($p \leq 0.4$) appear computationally tractable [22]. A *Variable Interaction Graph*[2] (VIG) is typically used to model variable interaction [87]. Every variable is represented by a vertex in VIG and an edge is established between two vertices if the two corresponding variables interact. Given a CNF formula $F$ with $m$ clauses and the length of clauses is bounded by $k$, VIG can be obtained in $m \times 2^k$ time, by extracting at most $2^k$ pairwise interactions from each of the $m$ clauses [88].

### 2.3.1 Variable Interaction Graph Visualization

The first step toward analyzing the VIG is visualization. Sinz introduces DPvis, a tool that initiates the study on VIG for explaining the difference performance of SS solvers between uniform random instances and application instances [1, 89, 90]. Sinz suggests VIGs to be laid out using a force-directed graph drawing algorithm by Fruchterman and Reingold [91] that are known to reflect graph clustering and symmetry. Fruchterman and Reingold's graph layout algorithm runs in polynomial time, $O(V+E)$, where $V$ is the number of vertices and $m$ is the number of edges in the given VIG.

After assigning only three decision variables, Sinz makes two observations that links to the performance of SS from the remaining VIGs solvers: 1) the remaining VIGs of easy application instances are greatly simplified and sometimes the initial connected VIG graph decomposes into several connected components; the remaining VIGs of hard instances instead exhibit a lower reduction rate and resemble the shape of original VIGs. 2) On an easy application instance, the branching decision has major impacts on both the size and the VIG of the remaining formula. The observations made by Sinz resonate with the weak backdoor concept, which suggests that on instances with small weak backdoors, selecting the right set of variables and setting them correctly greatly simplifies the problem instance.

[2] The graph is also referred as Variable Incidence Graph with the same acronym "VIG".

### 2.3.2 Complex Network Analysis

Besides intuitive visualization, well established methodologies from Complex Network research field have been brought to perform principled quantitative analyses on VIGs of SAT instances. A Complex Network is a network (graph) with non-trivial topology features that do not occur in simple graphs like random graphs, but which often occur in graphs modeling real systems such as the World Wide Web (WWW) and Social Networks [92, 93]. Walsh pioneered applying complex network analysis to graph representation of combinatorial search problems including graph coloring, timetabling and quasigroup problems [94, 95].

10 years after Walsh's seminal work [94], Ansótegui, Bonet, and Levy are the first that we are aware of to leverage complex network analysis methods in SAT domain, discovering the existence of scale-free structures in many application SAT instances [27]. The scale-free structures are not found in uniform random instances. The *Scale-free* structure is initially proposed by Albert, Jeong, and Barabási to model the topology of WWW. It is characterized by the power law distribution (i.e., $p(k) \sim k^{-\alpha}$) of vertex degrees of a complex network [96]. Applying the concept to graph representations of SAT instances, a scale-free structure suggests that variable frequencies of application instances follow a power law distribution. Implied by its name, scale-free graphs exhibit some kind of *self-similarity*, meaning that removing some vertices results in a similar graph but at a smaller scale. Interestingly, the scale-free structure is *preserved* even during the execution of a SS solvers, as shown in Figure 2.1. This result resonates with the observation made by Sinz through visualization, that hard application instances exhibits self-similarity [1]. Apart from showing the existence of scale-free structure in applications, Ansótegui, Bonet, and Levy do not address the connections between the scale-free structure and instance hardness.

Ansótegui, Giráldez-Cru, and Levy continue applying complex network analysis to SAT instances to discover another structural property, Community Structure, which can be found in application instances but not in uniform random instances [97]. Community structure concept generalizes the concept of connected components [98] by allowing (a few) connections between com-

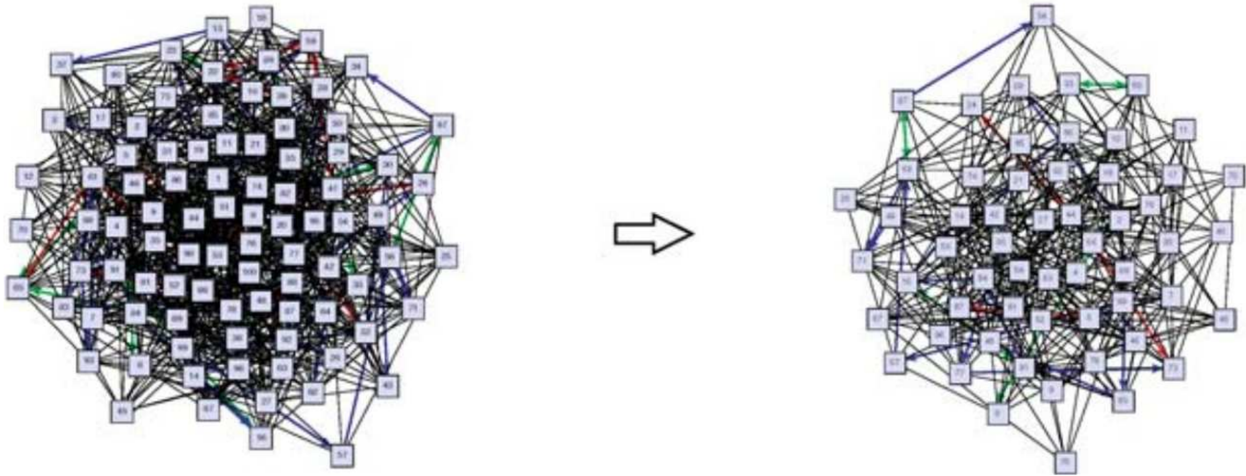


**Figure 2.1:** A uniform random 3SAT instance after three decisions in a systematic search [1].

munities. A graph has *community structure* if the graph divides naturally into groups (communities) of vertices with dense connections internally and sparse connections between communities.

*Modularity* is a metric evaluating the quality of community structure. Modularity is defined as the fraction of the edges that fall within the given communities minus the expected fraction if edges were distributed at random. The value of modularity is in the range $[-\frac{1}{2}, 1)$, with 1 indicating the strongest community structure [99]. Since the quality of community structure is subject to the partition of vertices, the modularity of a graph is defined as the *maximal modularity* for all possible partitions of the vertices [99]. Even though the decision version of modularity maximization is NP-Complete [100], fast heuristics, in which the cost of each iteration is linear in the number of edges, for computing a lower bound on modularity are available and viable for large instances SAT [101].

Ansótegui, Giráldez-Cru, and Levy report several findings on how SS solvers affect community structures in application instances. First, 15 out of 16 families of application instances from 2010 SAT Race [3] have a clear community structure, as indicated by the high lower bounds (around 0.8) of modularity. In contrast, uniform random instances at phase transition have lower bounds of modularity at about 0.15. Second, the use of the preprocessor SATElite [102] preserves the high

[3]http://baldur.iti.uka.de/sat-race-2010/

modularity in application instances, and eliminate almost all the small isolated components. This suggests SATElite can reduce the size of an instance without disrupting the community structure. Third, 72% of the learnt clauses by a conflict-driven clause learning SS solver "picosat" [103] are constructed locally on one of the communities. Since learnt clauses are built from the list of assigned decision variables, it implies that branching heuristics in SS solvers tend to branch on variables that are within a community. The authors do not present any study on the correlation between instance hardness and modularity in [97].

The correlation between instance hardness and community structure is later studied by Newsham *et al.* in [104] [4]. They extend the previous work [97] in three ways. First, they find that the number of communities and modularity were more correlated with the running time of the popular SS solver MiniSAT [13] than traditional features like number of variables, clauses or the clause-variable ratio. However, there is no direct comparison with more recent features such as backbone [22] and backdoor [56]. Moreover, as pointed out by Mull, Fremont, and Seshia [105], community structure alone is not sufficient for fully predicting the performance of SS solvers.

Second, they report a strong correlation between the quality of a learnt clause, as indicated by Literal Block Distance (LBD) [106], and the number of communities the learnt clause connects. LBD is a new metric for the quality of learnt clauses. LBD score a learnt clause based on the number of distinct decision levels that it is involved. The smaller LBD indicates a learnt clause with higher quality. This allows LBD to "glue" decision variables and variables implied by unit probation together. In modern SS solvers, learnt clause are periodically deleted according to the quality measure, due to memory constraint and the efficiency concern associated with propagating a large set of clauses. By using LBD instead of the learnt clause activity as the quality measure, Glucose [106] improve over previous SS solvers, including MiniSAT [13] and picosat [103]. This improvement is achieved by implicitly exploiting the community structure using LBD and keeping the learnt clauses that stay within a few communities. The fact that learnt clauses that are localized

[4] The paper won the Best Student Paper Award at SAT 2014

to a small set of communities potentially enabling the solver to essentially *partition* the problem into many small set of communities and solve them one at a time.

Third, the authors describe a method of generating random instances with controllable modularity and find that the running time of MiniSAT is highest when modularity is 0.05 and 0.13. This observation in part agrees with the finding in [97]. According to [97], uniform random instances, which are hard for SS solvers, have modularity at around 0.15.

Most recent works [107, 108] seek ways to explicitly exploit community structures to boost the performance of SS solvers. Inspired by the strong correlation between learnt clause quality (LBD) and the number of communities the learnt clause connects [104], Ansótegui, Giráldez-Cru, Levy, and Simon propose to use community structure to detect relevant learnt clauses [107], therefore proactively avoiding destroying community structures with learnt clauses. They propose a preprocessor *modprec*, which employs SS solvers to solve subformulas that contains pairs of connecting communities and insert the learnt clauses into the original formula, for the purpose of preventing learnt clauses being constructed across multiple communities during the actual solving. With the added community-respecting learnt clauses, two top SS solvers in SAT Competitions, glucose [106] and MiniSAT-Blvd [109] are further improved on instances from the application track of SAT competitions, taking the overhead of modprec into account.

### 2.3.3 Treewidth

The VIG also leads to another structural concept, treewidth (denoted as $tw$) [110], that is linked to instance hardness. It has been shown that SAT can be solved in time exponential only in treewidth using dynamic programming [111, 112]. It has two major implications. First, it presents an improvement over the worse-case complexity for traditional SAT solving, namely, $2^{tw} <= 2^n$. Second, if the treewidth of the VIG of an instance is bounded by a constant, the instance can be solved in polynomial time. *Treewidth* measures the "tree-likeness" of a graph $G$ and is defined over

Figure 2.2: A graph with eight vertices (left subfigure), and an optimal tree decomposition of it onto a tree with six nodes (right subfigure). $Treewidth = 3 - 1 = 2$.

a Tree Decomposition of $G$. A *Tree Decomposition*[5] is a mapping from $G$ to a tree $T$ that satisfies three conditions:

1. Every vertex of $G$ is in some tree node;
2. any pair of adjacent vertices in $G$ should be in the same tree node;
3. the set of tree nodes containing a vertex $v$ in $G$ forms a connected subtree.

The *width* of a tree decomposition is maximum size of tree node minus one. The *treewidth* of $G$ is the minimal width over all its possible tree decomposition. Figure 2.2 illustrates an optimal tree decomposition of a graph. The exact treewidth is NP-Hard to compute [114] and is also NP-hard to approximate with any constant factor [115]. Both polynomial-time heuristics for approximate treewidth computation [116] and efficient complete algorithms for exact treewidth computation [117–119] are available. The fastest time bound for computing exact treewidth is $O(1.8899^n)$, due to Fomin *et al.* [118]. Moreover, given that treewidth is bounded by a constant, exact treewidth computation can be done in quadratic time by Robertson and Seymour [110] and even in linear time by Bodlaender [120].

[5]Tree decomposition is also called junction trees, clique trees, or jointree in machine learning literature [113].

Even though a dynamic programming approach has been shown to solve SAT in time exponential only in treewidth, Mateescu [121] reports that treewidth is not a good indicator for practical running time of the SS solver "precosat" [122] on instances from the application track (which precosat won) of the SAT competition 2009. This observation is not surprising, since treewidth is a good measure for dynamic programming type algorithms (e.g., ones that fully traverse a tree decomposition of the problem), and most modern SS solvers do not exploit the tree decomposition.

Treewidth also finds various applications in accelerating SAT solving. One important concern when using tree decomposition is whether the instance has a small treewidth. One would expect that instances with small treewidth and yet hard for modern SS solvers should be suitable for the application of tree decomposition guided heuristics.

Bjesse *et al.* present empirical evidence that there are application instances with small treewidth [123]. The Dubois family of UNSAT instances [6], where the number of variables ranges from 150 to 6000, all have a constant treewidth of 4. Other application instances with 253 to 4566 variables have treewidth from 18 to 170. Bjesse *et al.* employ tree decomposition to guide variable ordering and conflict clause generation such that construction of clauses is restricted within one tree node. Despite the somewhat large treewidth, the tree-based approach method decreases the number of necessary decisions by one or more orders of magnitude. However, on one large instance, the tree-based method is about $4\times$ slower than the base SS solver. The authors conjecture that it becomes harder for the tree decomposition engine to find a high quality decomposition as the problem size increases, although there are other larger instances that the tree-based method performs better. Moreover, the authors do not reveal the overhead for tree decomposition. Finally, comparing the two tree-based method and traditional SS solver in terms of number of decisions is also unfair, since traversing the tree decomposition is likely to induce extra runtime overhead.

Huang and Darwiche employed a dtree, a bottom-up binary tree decomposition with each leaf node corresponding to a single clause [124], to compute the *semi-static* group variable ordering [125].

---

[6] Originally contributed by Olivier Dubois to the DIMACS collection, http://www.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/DIMACS/DUBOIS/descr.html.

Their group branching heuristic *statically* organizes variables into groups (each group corresponds to one tree node) according to dtree, and variables within the same group are *dynamically* ordered. The authors demonstrate that, by employing the group branching heuristic, a SS solver produces a divide-and-conquer behavior in which the instance is recursively decomposed into smaller problems that are solved independently. They show experimentally that integrating the group branching heuristic into the SS solver ZChaff [12] significantly improves performance in terms of CPU time on a range of benchmark problems that exhibit structure. The overhead of dtree computation is small and beneficial supported by the substantial speedup in the actual SAT solving. For instance, the largest CPU overhead is 66 seconds and tree-based dtree-ZChaff takes 4206 seconds, whereas the base solver ZChaff takes 28495 seconds.

Li and Beek extends Huang and Darwiche's work [124] with a fully dynamic branching heuristic with on-the-fly tree decomposition along with SAT solving [126]. Their dynamic branching heuristic is inspired by the observation made by Sinz [89] that the VIG changes drastically over the course of SAT solving. Their dynamic branching heuristic take a top-down approach. It dynamically computes the *separator variables*, the variables that decompose the graph, based on the remaining formula. The remaining formula is updated every time a decision variable is assigned by removing the assigned decision variables and the implied variables assigned by unit propagation, as well as the clause satisfied by decision variables and implied variables. This way, a full tree decomposition is not needed, and the decomposition always consider the latest VIG. An empirical study on instances from application track of SAT competition 2002 demonstrates the dynamic branching heuristic can reduce the number of decisions needed to solve an instance. In terms of CPU time, ZChaff with dynamic ordering outperform the original ZChaff on 7 of 13 instance families, and outperform Dtree-Zchaff on 8 of 12 instance families.

Habet, Paris, and Terrioux [127] further advance *static* tree decomposition guided SAT solving by learning "goods" (partial assignments that lead to a satisfying assignment) and "nogoods" (partial assignments that lead to a conflict) for separator variables. Similar to [125], Habet, Paris, and Terrioux utilize a static tree decomposition to generate a variable ordering. Tree decomposition

guided satz [54] outperforms the base SS solver satz on 10 out of 21 instance families. The result might seem unremarkable. However, it can outperform all the other state-of-the-art SS solvers (at that time) including MiniSAT [13], Rsat [128], and ZChaff [12]. Process saving with the static tree decomposition is straightforward, but how to perform process saving in the context of dynamic decomposition (see for example [126]) remains an open question.

In the face of large application instances, Monnet and Villemaire propose a scalable tree decomposition [129]. The complexity of the scalable tree decomposition heuristic is $O(log(n) * (n + m))$, where $n$ is the number of variables and $m$ is the number of clauses. In practice, Monnet and Villemaire's tree decomposition heuristic scales to application instances from various SAT competitions with hundreds of thousands of variables and millions of clauses, and the computation can be done in less than 0.25 second, while the other decomposition methods [124, 130] run out of memory on most instances. Empirical data also demonstrate that the scalable decomposition leads to encouraging improved performance of MiniSAT [13]. On most of the instances from four application domains, the tree-based MiniSAT performs the original MiniSAT by a large margin. The tree-based MiniSAT solves one of the bounded model checking instance ($n = 118,426$ and $m = 375,699$) in 0.42 seconds, whereas the original MiniSAT runs out of time.

### 2.3.4 Summary

While applying apriori structural analysis to variable interaction topology to understand how SS solvers works and improve the performance of SS solvers has been a highly fruitful topic, we are not aware of any existing work that applies similar techniques to SLS solvers. How to leverage complex network inspired structures for SLS solvers remains an open question.

## 2.4 SLS Search Space

Previous structural properties are *instance-specific*, in the sense that they characterize an instances itself, independent of the methods that are employed to solve it [7]. In this section, we instead review the structural properties obtained from posteriori analysis that are unique to SLS solvers; they characterize how SLS solvers "see" a problem instance. An SLS solver employs an objective function[8] to rank states (a complete assignment to variables of the formula), and picks a "neighboring" state by flipping a variable in the current state that maximizes the improvement to the objective function [131]. The objective function for a CNF-based SLS solver for unweighted MAXSAT (take WalkSAT [15] for example) is typically the number of unsatisfied clauses, while a global optimum is a state where the objective function evaluates to zero, i.e., all clauses are satisfied. SLS solvers frequently encounter a sequence of states in which it is impossible to reduce the number of unsatisfied clauses. Moves through these regions, called *plateau moves*, usually dominate the running time of SLS solvers [132]. Analyzing the search space of a SAT instance is critical for developing more efficient local search operators. Enumerating the entire exponential search space is prohibitively expensive; the problem within the reach of enumeration is only up to 50 variables [133]. Exhaustive enumeration of the space space is also unnecessary, since not all part of search space are equally hard or interesting. More specifically, the part of search space that contains no plateau is easy, and the part of search that contains only states of high evaluations (large number of unsatisfied clauses) is uninteresting. Researchers therefore identify search space features that impact the performance of SLS solvers.

Frank, Cheeseman, and Stutz present the first study into the search space of SAT problems from the perspective of plateaus [134]. Plateau moves represent a balanced act. On one hand, continuing to explore a plateau that contains exits can be rewarded with a better state. On the other hand, giving up on a plateau that is large and yields no exits can save substantial computational

[7]Strictly speaking, the original backdoor concept is subject to the subsolver being used. Nevertheless, we can also view backdoor as the intrinsic shortcut to solving an instance, for the sake of unity.

[8]Objective function is also called "evaluation function" or "cost function".

resources for other more fertile plateaus. The authors present a systematic study of plateaus on three randomly generated instance families (SAT, UNSAT, and Cluster). To cope with the exponential size search space, the authors employ GSAT [14] to samples states. Several findings are reported by the authors. First, local optima (plateaus without exits) tend to be small but occasionally may be very large. Second, plateaus with exits, called *benches*, tend to be much larger than local optima, and some benches have very few exit states which local search can use to escape. The first two findings suggest that it might not be a good idea to explore plateaus looking for exits, at least on the three studied instance families. Third, local optima can be escaped without unsatisfying a large number of clauses. This suggests a complete restart might not be necessary. Finally, global optima of randomly generated problem instances form clusters, which behave similarly to local optima. The last finding indicates that ideas like tunneling between local optima might be able to effectively "jump" between the clusters of local optima by recombining local optima [135, 136]. Experiments conducted by Zhang [48] agrees that local minima from WalkSAT [15] form large clusters, and their search space constitute big valleys, where high quality local optima typically share large partial solutions with optimal solutions. The tunneling idea has also been shown to be useful in Traveling Salesperson Problem [137] and NK-Landscape [138], and we are not aware of any work that apply this idea to SAT domain. Apart from the interesting observations, Frank, Cheeseman, and Stutz do not demonstrate how the plateau features impact the practical performance of SLS solvers. In addition, the observation on random instances might not generalize to application instances, on which SLS solvers struggle.

Hoos [139] presents an empirical study on the connections between the performance of the SLS solver WalkSAT [15] and search space features, on the two SAT-encodings (sparse and compact) of two respective NP-problems: Random Binary Constraint Satisfaction Problem and Random Hamilton Circuit Problem. Three search space features are studied by Hoos: 1) solution density; 2) standard deviation of objective function (denoted as *sdnclu* in [139]); and 3) the number of neighbors with the same evaluation (denoted as *blmin* in [139]). *sdnclu* and *blmin* characterize the overall ruggedness of the search space; a rugged search space offers more gradient for SLS solvers

to follow, and is expected to be easier. Through the comparison of the performance of WalkSAT between the two encodings of the NP-Hard problems, the author finds that the search space induced by the sparse encoding is easier for WalkSAT to navigate, even though the compact encoding yields lower solution density achieved by reducing the number of variables. This (a bit counter-intuitive) observation can be explained by the ruggedness of the search space, as indicated by *sdnclu* and *blmin*. The search space of the compact encoding has significantly lower *sdnclu* and substantially higher *blmin*, which indicates a flatter search space. However, the author does not present any utility of the search space features in boosting the performance of SLS solvers, and leaves it as an open research question. Hoos, Smyth, and Stützle also study autocorrelation [140] and fitness distance correlation [141] as instance hardness measures on unweighted and weighted random MAX-SAT problem, and find fitness distance correlation to be a better measure [142]. Nonetheless, we are not aware of any work that employ the two features for explaining instance hardness in SAT domain.

Schuurmans and Southey propose three characteristics for the effectiveness of SLS solvers, then demonstrate substantial correlations with the performance of SLS solvers in terms of number of bit flips, and finally employ the characteristics to drive the design of a new SLS solver SDF that conform the performance predictions made by the characteristics. [143, 144][9]. The three characteristics are depth, mobility and coverage. *Depth* measures how many clauses remain unsatisfied as the search proceeds, and it indicates how "deep" the local search is in the space. *Mobility* measures how rapidly a local search moves in the space, and it is computed by calculating the Hamming distance between states that are sampled from every $k$ steps. *Coverage* measures how systematically the local search explores the entire space, and a *rate* of coverage is estimated by how fast is the gap between two most distant states (the gap is conceptually similar to graph diameter [145]) being reduced. The authors show that the poor indication under any one of the characteristics leads to poor performance in overall SAT solving, i.e., *necessity*. The authors further demonstrate that simultaneously good depth, mobility, and coverage scores is *sufficient* to ensure that overall effective problem solving performance is obtained, independent of other algorithmic details. Driven

[9]The conference paper won the Outstanding Paper Award in AAAI'2000.

**Table 2.1:** Summary of Structural Properties. "Inst-Spec" indicates whether the structural property is instance-specific. "NP-Hard" indicates whether acquiring the exact structural property is NP-Hard. "Heuristic" indicates whether polynomial-time heuristics is available. "SS Perm" (or "SLS Perm") indicates weather the structural property can be used to explain the performance of SS (or SLS) solvers. "Impr SS" (or "Impr SLS") indicates weather the structural property has been exploited to improve the performance of SS (or SLS) solvers.

| Struc Prop | Inst Spec | NP-Hard | Heuristic | SS Perm | Impr SS | SLS Perm | Impr SLS |
|---|---|---|---|---|---|---|---|
| Backbone | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| Backdoor | Yes | Yes | Yes | Yes | Yes | No | No |
| Var Dep | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| VIG Viz | Yes | No | N/A | Yes | No | No | No |
| Com Struc | Yes | Yes | Yes | Yes | Yes | No | No |
| Treewidth | Yes | Yes | Yes | Yes | Yes | No | No |
| Search Space | No | N/A | N/A | No | No | Yes | Yes |

by the proposed characteristics, the authors design a new SLS solver, SDF, which employs a smoothed version of the standard GSAT [14] objective function to obtain better depth scores, and simultaneously uses multiplicative clause weight updating to obtain better mobility and coverage scores. The improved characteristic scores of SDF are confirmed empirically, which in return leads to effectiveness improvement in terms of number of bit flips over other SLS solvers including the best SLS solver at the time DLM [146] and Novelty+ [147], on a range of instances. However, the improvement in number of bit flips does not translate into an advantage in raw solving time, because SDF is consistently a factor of three to four times slower than DLM in flipping one bit.

## 2.5 Identifying a Gap in Previous Works

The structural properties we reviewed are summarized in Table 2.1. We can identify several interesting trends in Table 2.1. First, most structural properties are *instance-specific*, meaning that they reveal certain aspects of a problem instance's characteristics. Second, even though most instance-specific structural properties are NP-Hard to acquire exactly, heuristics are available. Third, all instance-specific structural properties can help explain and understand the performance of SS solvers, while only backbone and variable dependency are used to explain the performance of SLS solvers. Finally, most instance-specific structural properties can be utilized to improve SLS solvers, whereas only backbone and variable dependency have been exploited by SLS solvers before.

In addition to the studies on structural property that are well classified in our taxonomy, more recent works tend to bridge between different structural properties. Zabiyaka and Darwiche [148] point out that when variable dependence is present, problems with high treewidths can still be solved efficiently. Ansótegui *et al*. [149] establish a new problem hardness measure based on backdoor and treewidth, while Gaspers and Szeider [150] study the size of backdoor, given that treewidth is bounded. Newsham *et al*. [35] present a visualization of how a popular branching heuristic selects branching variables with respective to the community structures.

We have conducted a synthesis of large number of existing works, which enables us identify a gap in previous work. Recall that backbone, backdoor, variable dependency, variable interaction, community structure and treewidth are all instance-specific structural properties. These instance-specific structural properties reveal the characteristics of an instance in many different aspects. In contrast to the enormous existing works that investigate the way instance-specific structural properties impact the performance of SS solvers and demonstrate that the instance-specific structural properties can be leveraged to improve the performance of SS solvers, rare existing works have exploited the instance-specific structural properties for SLS solvers. We are only aware of three works [48, 72, 79] that explicitly exploit instance-specific structural properties of instances to improve the performance of SLS solvers. Linking SLS solvers that are oblivious to instance-specific structural properties with the fact that SLS solvers fall far behind SS solvers on application instances, we conjecture that SLS solvers can be improved through exploiting instance-specific structural properties.

The success of the only three structure-aware SLS solvers and the abundance of instance-specific structural properties have inspired our proposed work in the following way. Chapter 3 studies the decomposability of VIGs with pseudo backbone constructed from good local optima. Chapter 4 demonstrates the utility of the decomposition in improving SLS solvers. Chapter 5 explores alternative representations that automatically respect variable dependencies defined by arbitrary Boolean functions. Empirical studies demonstrate that the advantage of the alternative representations over the traditional CNF representations.

# Chapter 3

# Decomposing Variable Interaction Graphs with Pseudo Backbones

The *decomposability* of SAT instances have been extensively exploited with success by SS solvers [107,108,123,125–127,129] (see Subsection 2.3.2 and Subsection 2.3.3 for detailed reviews). Community structures [104] and treewidth [150] are two major structural properties that can be used to quantify the decomposability of a graph. Meanwhile, SLS SAT solvers direct which variables to flip using only the objective function, and are completely oblivious of the decomposability of application SAT instances that is inherent to the original problem domain. Combining the absence of decomposibility exploiting strategies in SLS solvers with the fact that SLS solvers are dominated by SS solvers on application instances, we conjecture that SLS solvers can be improved by exploiting the decomposability of application instances. In this chapter, we propose a first step toward exploiting decomposability with SLS solvers.

Our proposed work is two-fold. First, we study the feasibility of decomposing SAT instances in the context of SLS solvers. Decomposing SAT instances with SS solvers is straightforward, since an SS solver assigns one decision variable at a time, and the assigned variables and variables implied through unit propagation can be used to simplify the Variable Interaction Graphs (VIGs) of instances, which naturally leads to decomposition. On the contrary, SLS solvers search in the space of tentative assignments to all variables. How to "fix" variables so that a VIG can be decomposed in the context of SLS solvers is non-trivial. Inspired by Zhang's success in guiding SLS solvers with pseudo backbones extracted from good local optima [48] and by the definition that backbone variables are *fixed* variable assignments across all global optima [22], we propose to decompose VIGs with pseudo backbones sampled by SLS solvers. We then propose two SAT-specific optimizations that lead to better decomposition than on general pseudo Boolean optimization problems. Our empirical

study suggests that pseudo backbones can vastly simplify SAT instances, which further results in decomposing the instances into thousands of connected components.

Second, we demonstrate the *utility* of the decomposition obtained from pseudo backbone, in improving SLS solvers. Our work is motivated from two lines of prior works. One line of prior works suggests that local optima and global optima appear to form clusters. This observation has been reported independently in different contexts by Frank, Cheeseman, and Stutz in [134] on search space analysis, by Parkes in [52] on the impact of large backbones on the performance of SLS solvers, by Zhang in [48] on pseudo backbone guided SLS solvers, and by Qasem and Prugel-Bennett in [151] on MAX-SAT fitness landscapes. The other line of works suggests tunneling through local optima can be achieved using partition crossover [88, 135, 136]. *Partition Crossover* fixes pseudo backbone variables (i.e., variable assignments shared between two local optima) to decompose the VIG into independent connected components, and then recombines partial solutions to different components, for the purpose of effectively "jumping" through the clusters of local optima. Partition crossover has been shown to be useful in Traveling Salesperson Problem [137] and NK-Landscape [138], and we are not aware of any work that apply this idea to SAT domain. We propose to fill this gap by applying partition crossover to exploit the decomposability obtained from pseudo backbone. Our finding serves as a key stepping stone for applying the powerful recombination operator, partition crossover, to SAT domain.

## 3.1 Feasibility of Decomposing VIGs with Pseudo Backbones

Our goal in this chapter is to demonstrate the feasibility of decomposing VIGs with pseudo backbones. We conjecture that good local optima sampled by an SLS solver will share some common variable assignments, i.e., pseudo backbones are non-empty. Moreover, the pseudo backbone can be used to simplify VIGs of application instances by removing assigned pseudo backbone variables as well as the clauses satisfied by assigned pseudo backbone variables, and possibly decomposing the VIGs.

We will approach the goal by taking the following three steps. First, we will start by identifying a selected set of application instances that demonstrate potential decomposability, namely, having small treewidths. Our initial pool of application instances include all 299 satisfiable instances from both hard combinatorial track and industrial track in SAT Competition 2014[10], where the hard combinatorial track contains 150 satisfiable instances and the industrial track contains 149 satisfiable instances[11]. Second, we run one of the best performing SLS solver on application instances [152], AdaptG$^2$WSAT [16], on the selected instances to collect good local optima at low evaluation level, and then extract shared variable assignments from the local optima to construct pseudo backbones. Finally, we further propose two SAT specific optimizations that lead to provably better decomposition than on general pseudo Boolean optimization problem, and evaluate how well the pseudo backbones decompose the VIGs, by comparing the original VIGs with the simplified VIGs from two aspects: the visualization of the remaining VIGs, the number of connected components.

### 3.1.1 Identifying Application Instances with Potential Decomposability

Preprocessors like the well-known SATElite [102] and the more recent Coprocessor [153] are commonly used by SS solvers to simplify an instance by reducing the number of variables and the number of clauses, which leads to smaller search space and more efficient unit propagation. The impact of preprocessing on the treewidths of application instances is unknown, which leads to Question 1.

**Question 1.** *Can a preprocessor reduce the treewidth of application instances?*

[10]http://www.satcompetition.org/2014/

[11]The original benchmark set in the industrial track contains 150 satisfiable instances. However, openstacks-sequencedstrips-nonadl-nonnegated-os-sequencedstrips-p30_3.085-SAT.cnf and openstacks-p30_3.085-SAT.cnf are identical instances, despite their different names.

To Question 1, we compute the treewidths for 299 satisfiable application instances before and after being preprocessed by SATElite[12]. The treewidths are computed using the **treed** library authored by Subbarayan [13]. **treed** library provides a C++ implementation of several tree decomposition heuristics, including Maximum Cardinality Search, Minimum Degree, Minimum Fill-in. The details about the heuristics can be found in [116]. The Minimum Degree heuristic is used in our experiments, because it appears to scale the best among the all heuristics implemented in **treed** on large graphs with up to millions of vertices. The time spent on tree decomposition is limited to be one hour. All experiments in this proposal are run on HP XW6600 workstations each equipped with two Intel Xeon E5450 processors running at 3.0GHz and 16GB of memory.

Despite using the most scalable tree decomposition heuristic Minimum Degree, 20 out of 149 industrial instances cannot finish tree decomposition within the one hour limit. Applying the preprocessor reduces the number of timed-out industrial instances by one. The preprocessor further solves 2 industrial instances without branching on any decision variable, i.e., the 2 instances have both a backdoor of size zero. This leads to 128 instances where we have valid treewidths on both the original instances and the preprocessed instances. The comparison between the treewidths of 128 industrial instances before and after preprocessing is reported in Figure 3.1. Figure 3.1 suggests that applying the preprocessor can reduce treewidths of industrial instance by a notable margin, with the median of treewidths reduced from 815 to 718.

On the 150 original hard combinatorial instances, only 10 of them cannot finish tree decomposition within one hour. Applying the preprocessor again eliminates one of the timed-out instance, and it further solves 11 more instances. On the remaining 130 instances, their per instance comparison in terms of treewidth is reported in Figure 3.2. On the hard combinatorial instances, Figure 3.2 suggests that preprocessing has minimal impact on the treewidths, which is also confirmed by the fact that the median treewidths before and after be preprocessed are almost the same (290 versus

[12]In this proposal, we use the SATElite implementation that is built into MiniSAT 2.2. http://minisat.se/MiniSat.html

[13]http://www.itu.dk/people/sathi/treed/

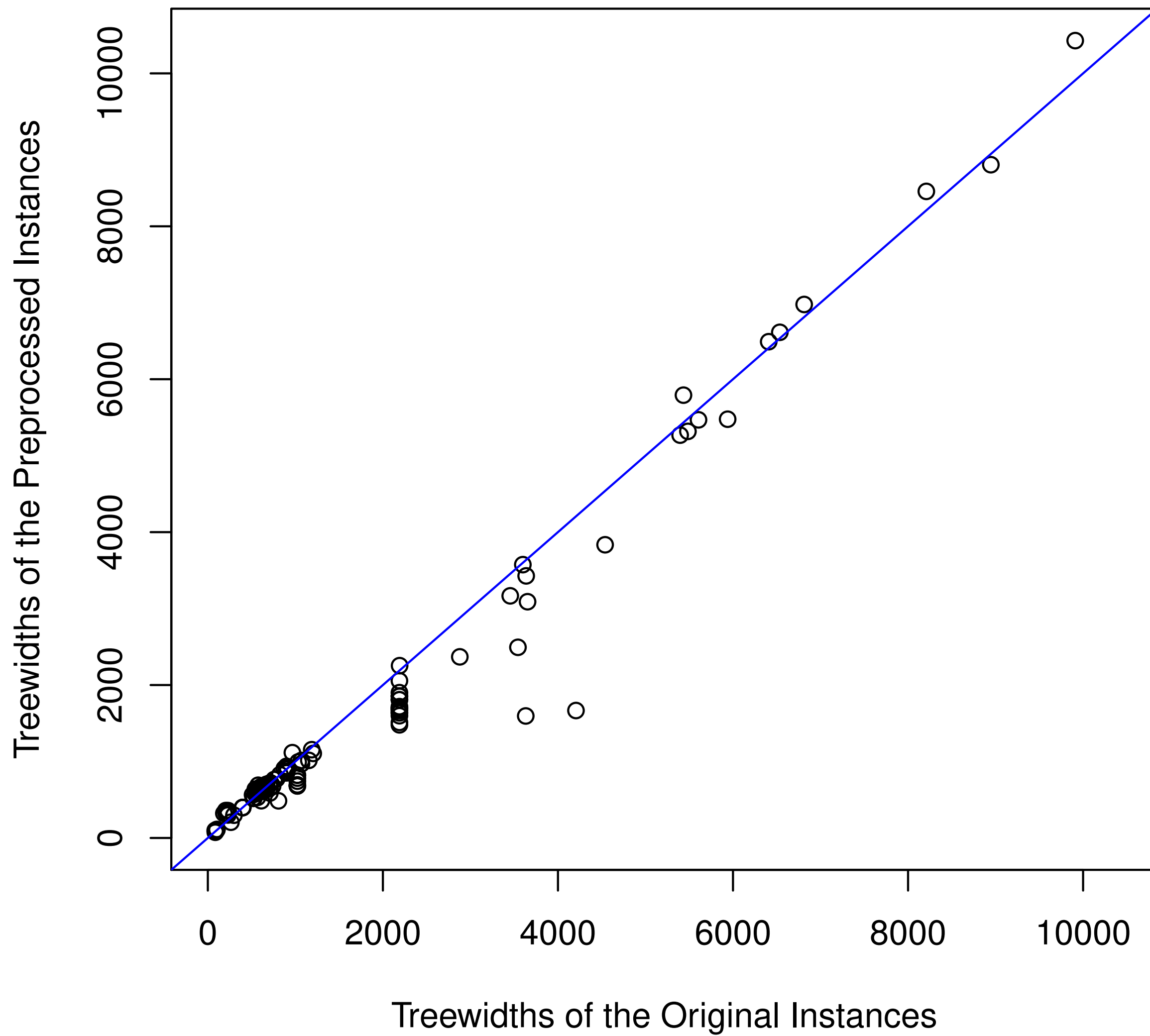



**Figure 3.1:** The impact of preprocessing on the treewidths (computed by the Minimum Degree heuristic) of 128 satisfiable instances from industrial tracks of SAT Competition 2014. The diagonal line is the $y = x$. Each point represents a pair of treewidths of an original instance and the corresponding preprocessed instance. Points below (above) indicates the treewidth of the original instance is larger (smaller).

291). Nevertheless, preprocessing is still advantageous on hard combinatorial instances, because it solves a significant number (11) of the instances without branching.

Supported by the empirical analysis on the two sets of experiments, we have an empirical answer to Question 1: preprocessing can reduce and almost never increase the treewidths of application instances. We hence use the preprocessed instances instead of the original application instances for our future studies in this chapter. The set of 258 preprocessed application instances include 128 preprocessed industrial instances and 130 preprocessed hard application instances.

Treewidth quantifies the size of the most densely connected and indecomposable subgraph (i.e., size of the largest tree node), and can be used as a metric of the inherent decomposability of a graph [150]. Assigning all variables in any tree node guarantees to decompose the graph. A small treewidth suggests that assigning a small set of variables corresponding according to the tree decomposition can guarantee to decompose the graph. Meanwhile, we notice that there is a high variation in the size of preprocessed instances, i.e., the number of variables for the 258 preprocessed application instances varies vastly from 55 to 1,471,468 with an average of 47,724 and a standard deviation of 174,429. To accommodate the variation, we use normalized treewidth, i.e., treewidth to number of variables ratio (denoted as *tw.n.pre*), as a metric between zero and one for evaluating inherent decomposability for the preprocessed application instances. We therefore select application instances that are known to have small *tw.n.pre* (i.e., highly decomposable) as the first step demonstrating the feasibility of decomposing VIGs with pseudo backbones. In other words, if pseudo backbones can not decompose VIGs with small *tw.n.pre*, it would be even less likely to decompose VIGs with larger *tw.n.pre*.

To study the distribution of *tw.n.pre* in the preprocessed application instances, we plot the histogram in Figure 3.3. The histogram indicate that highest frequency of *tw.n.pre* is within the range from 0.00 to 0.01 (the first bar has a frequency of 28). This suggests there are 28 preprocessed application instances that can be decomposed when assigning at most 1% variables. We also observe that *tw.n.pre* for 199 out of the 258 application instances are less than 0.4. This is encouraging

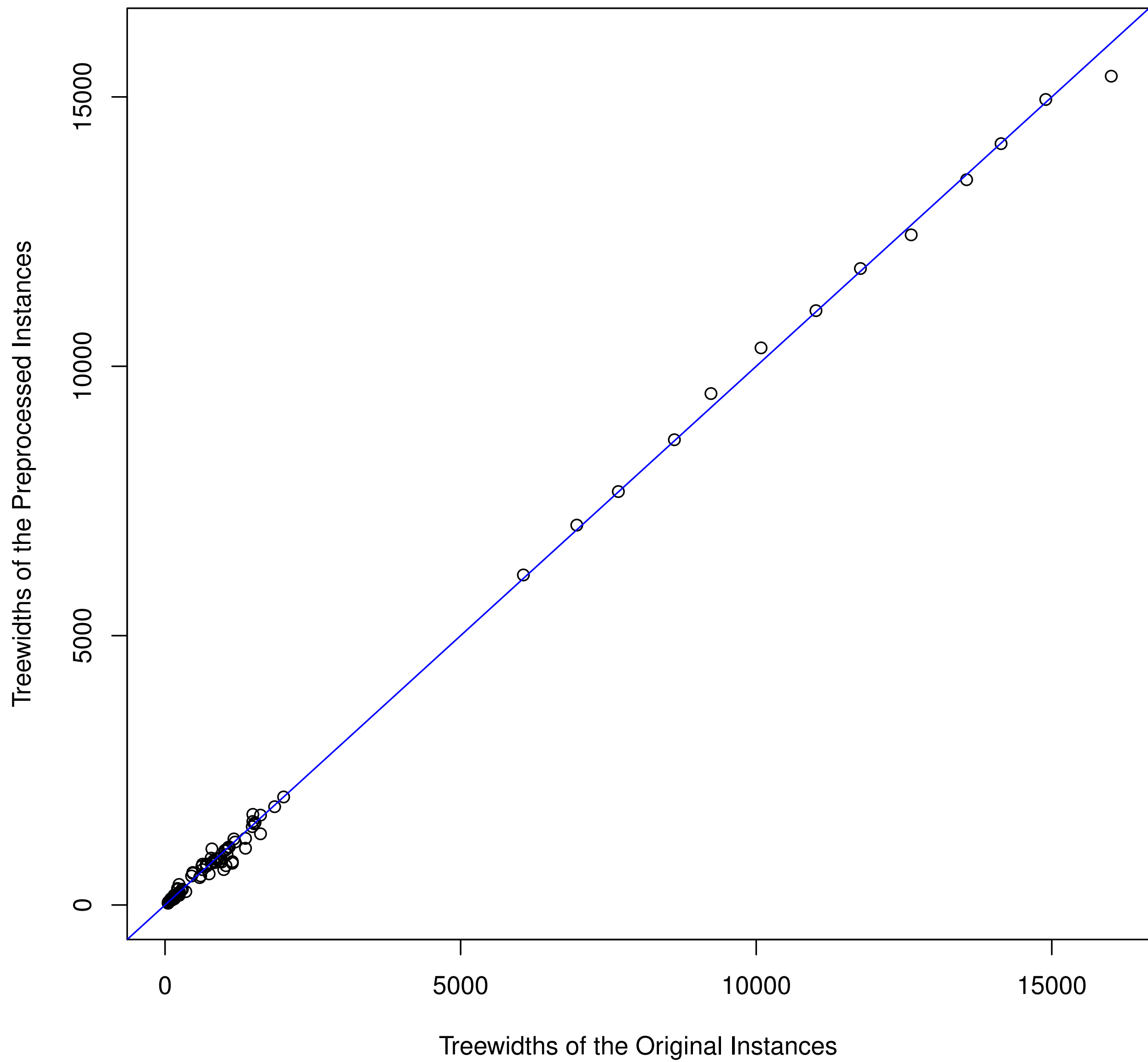



**Figure 3.2:** The impact of preprocessing on the treewidths (computed by the Minimum Degree heuristic) of 128 satisfiable instances from hard combinatorial tracks of SAT Competition 2014. The diagonal line is the $y = x$. Each point represents a pair of treewidths of an original instance and the corresponding preprocessed instance. Points below (above) indicates the treewidth of the original instance is larger (smaller).

because many real world application instances from a wide variety of domains are indeed highly decomposable.

We select the 28 preprocessed application instances with $tw.n.pre < 0.01$ as the testbed for exploring the feasibility of decomposition with pseudo backbones. We list the details on the selected instances in Table 3.1. We note that the 28 selected instances come from only 6 problem domains: LABS (Low Autocorrelation Binary Sequence), prime (semiprime factoring problem), atco (Air Traffic Controller shift scheduling problem), SAT-instance (autocorrelation in combinatorial design), AProVE (termination analysis of JAVA program), and aaai-ipc (planning). The instance with smallest *tw.n.pre* is LABS_n088_goal008, in which there are 182,015 variables, and yet its treewidth is only 203.

To illustrate how the variable interaction topologies of the 28 selected instances lead to the small *tw.n.pre*, we generate their VIGs using force-directed graph layout algorithm[14], following Sinz's methodology in [1]. We notice that, among the 28 selected instances, the VIGs of instances from the same class are visually similar. Therefore, one representative VIG is picked from each of 6 problem domains. The 6 representative VIGs are presented in Figure 3.4, Figure 3.5 and Figure 3.6, in which a red dot represents a variable and a black line between two red dots represents the two relative variables co-occur in some clause.

The presented VIGs illustrates the decomposability nature of the variable interaction topologies of the selected preprocessed application instances. atco_enc3_opt1_13_48 (Figure 3.4 top) consists of several linear topologies that loosely interleaves. LABS_n088_goal008 (Figure 3.4 bottom) exhibits a densely connected "core" at its center, and the connections become progressively sparser as being away from the core. SAT_instance_N=49 (Figure 3.5 top) appears axisymmetric with two cores on each side and the connections between the two cores are sparser. aaai10-ipc5 forms several layers of "clusters" with looser connections between clusters. The aforementioned 4 VIGs exhibits high visual decomposability. Surprisingly, prime2209-98 (Figure 3.5 top) is already separated into

[14] We use the Python package graph-tool (https://graph-tool.skewed.de/). Graph-tool is known for its efficiency on large graphs, due to its C++ core implementation.

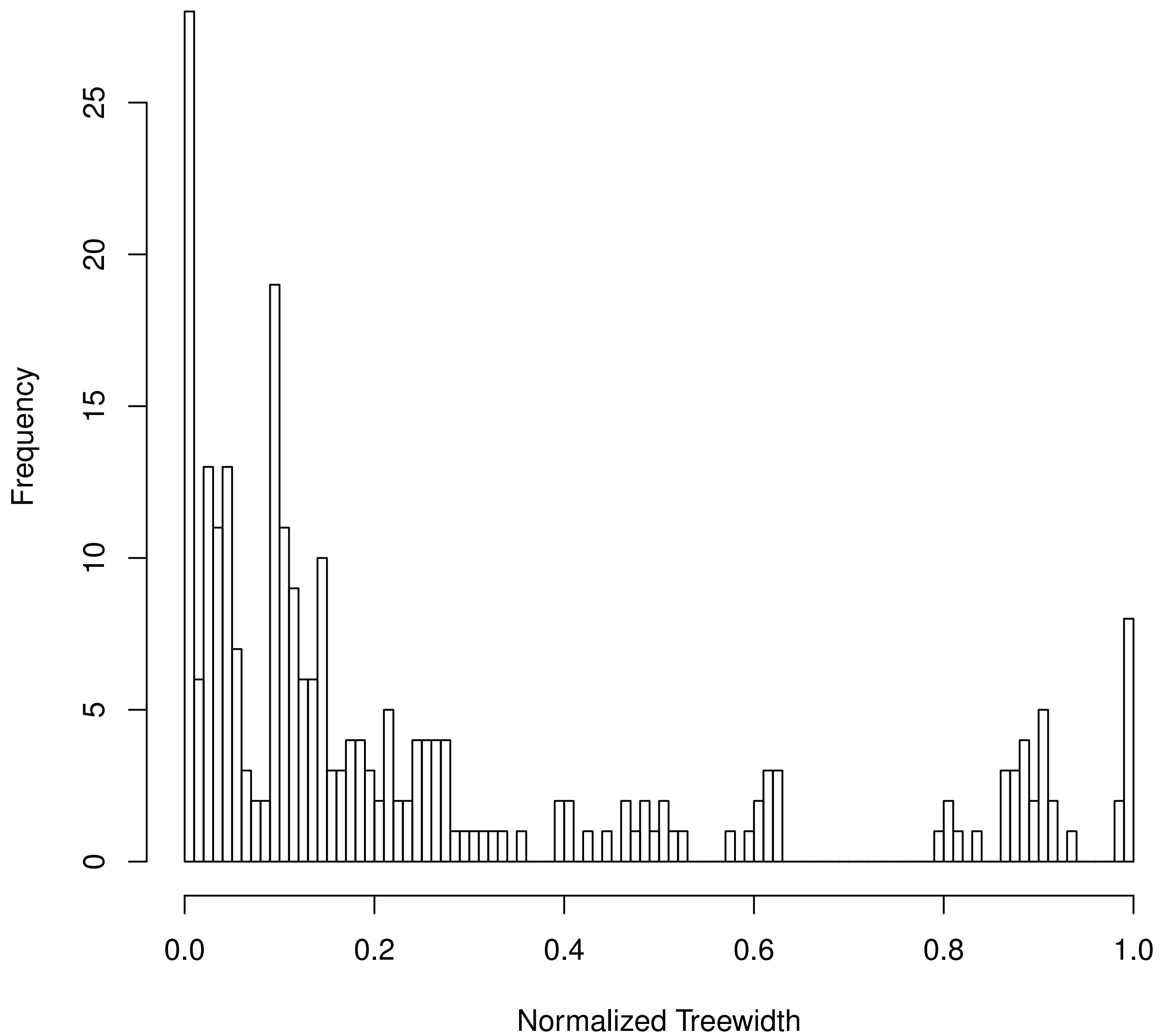
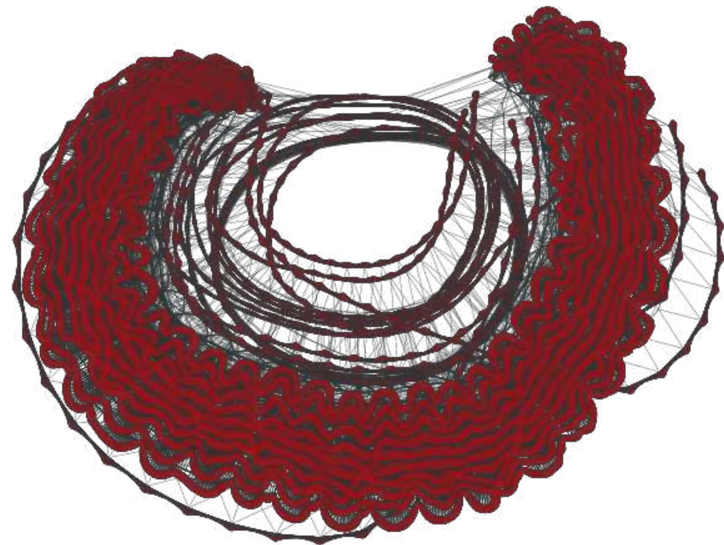



**Figure 3.3:** Normalized Treewidths of 258 preprocessed application instances. Each bar counts the frequency of the normalized treewidths being within an interval of size 0.01.

**Table 3.1:** Detailed statistics on the 28 selected preprocessed application instances, sorted by *tw.pre.n*. “inst” indicates the instance name. “track” indicates the track from which the application instance is: “comb” means hard combinatorial track; “indu” means industrial track. “n.pre” indicates the number of variables. “m.pre” indicates the number of clauses. “tw.pre” indicates the treewidth. “tw.pre.n” indicates the normalized treewidth.

| inst | track | n.pre | m.pre | tw.pre | tw.n.pre |
|---|---|---|---|---|---|
| LABS_n088_goal008 | comb | 182015 | 638771 | 203 | 0.0011 |
| LABS_n082_goal007 | comb | 147213 | 509579 | 192 | 0.0013 |
| LABS_n081_goal007 | comb | 138803 | 479962 | 186 | 0.0013 |
| LABS_n078_goal006 | comb | 118601 | 405972 | 179 | 0.0015 |
| prime2209-98 | comb | 16758 | 88592 | 33 | 0.0020 |
| LABS_n066_goal005 | comb | 71336 | 245401 | 146 | 0.0020 |
| atco_enc3_opt2_18_44 | indu | 1067657 | 4305314 | 2494 | 0.0023 |
| atco_enc3_opt2_05_21 | indu | 1293612 | 5193145 | 3090 | 0.0024 |
| atco_enc3_opt1_03_53 | indu | 991419 | 3964576 | 2369 | 0.0024 |
| atco_enc3_opt1_04_50 | indu | 1320073 | 5313526 | 3167 | 0.0024 |
| atco_enc3_opt1_13_48 | indu | 1471468 | 5921783 | 3575 | 0.0024 |
| LABS_n052_goal004 | comb | 34674 | 120036 | 115 | 0.0033 |
| LABS_n051_goal004 | comb | 32522 | 112980 | 116 | 0.0036 |
| SAT_instance_N=111 | comb | 72001 | 287702 | 269 | 0.0037 |
| SAT_instance_N=99 | comb | 57407 | 229153 | 239 | 0.0042 |
| SAT_instance_N=93 | comb | 49745 | 199060 | 221 | 0.0044 |
| LABS_n045_goal003 | comb | 22095 | 76929 | 102 | 0.0046 |
| SAT_instance_N=85 | comb | 41562 | 166062 | 203 | 0.0049 |
| LABS_n044_goal003 | comb | 20259 | 70752 | 100 | 0.0049 |
| AProVE09-06 | indu | 37726 | 192754 | 206 | 0.0055 |
| SAT_instance_N=77 | comb | 34129 | 136225 | 189 | 0.0055 |
| SAT_instance_N=75 | comb | 32356 | 129290 | 187 | 0.0058 |
| SAT_instance_N=69 | comb | 27286 | 109087 | 165 | 0.0060 |
| SAT_instance_N=72 | comb | 29289 | 117013 | 184 | 0.0063 |
| SAT_instance_N=63 | comb | 22780 | 90906 | 153 | 0.0067 |
| aaai10-ipc5 | indu | 308480 | 2910796 | 2254 | 0.0073 |
| SAT_instance_N=55 | comb | 17272 | 68977 | 145 | 0.0084 |
| SAT_instance_N=49 | comb | 13766 | 54806 | 125 | 0.0091 |



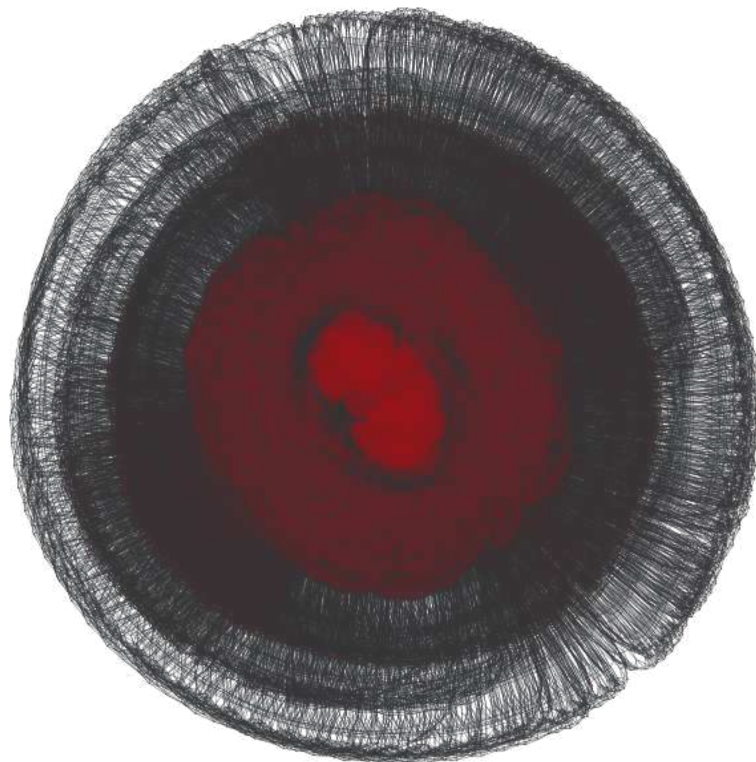


**Figure 3.4:** Variable interaction graphs of preprocessed application instances atco_enc3_opt1_13_48 (top) and LABS_n088_goal008 (bottom).

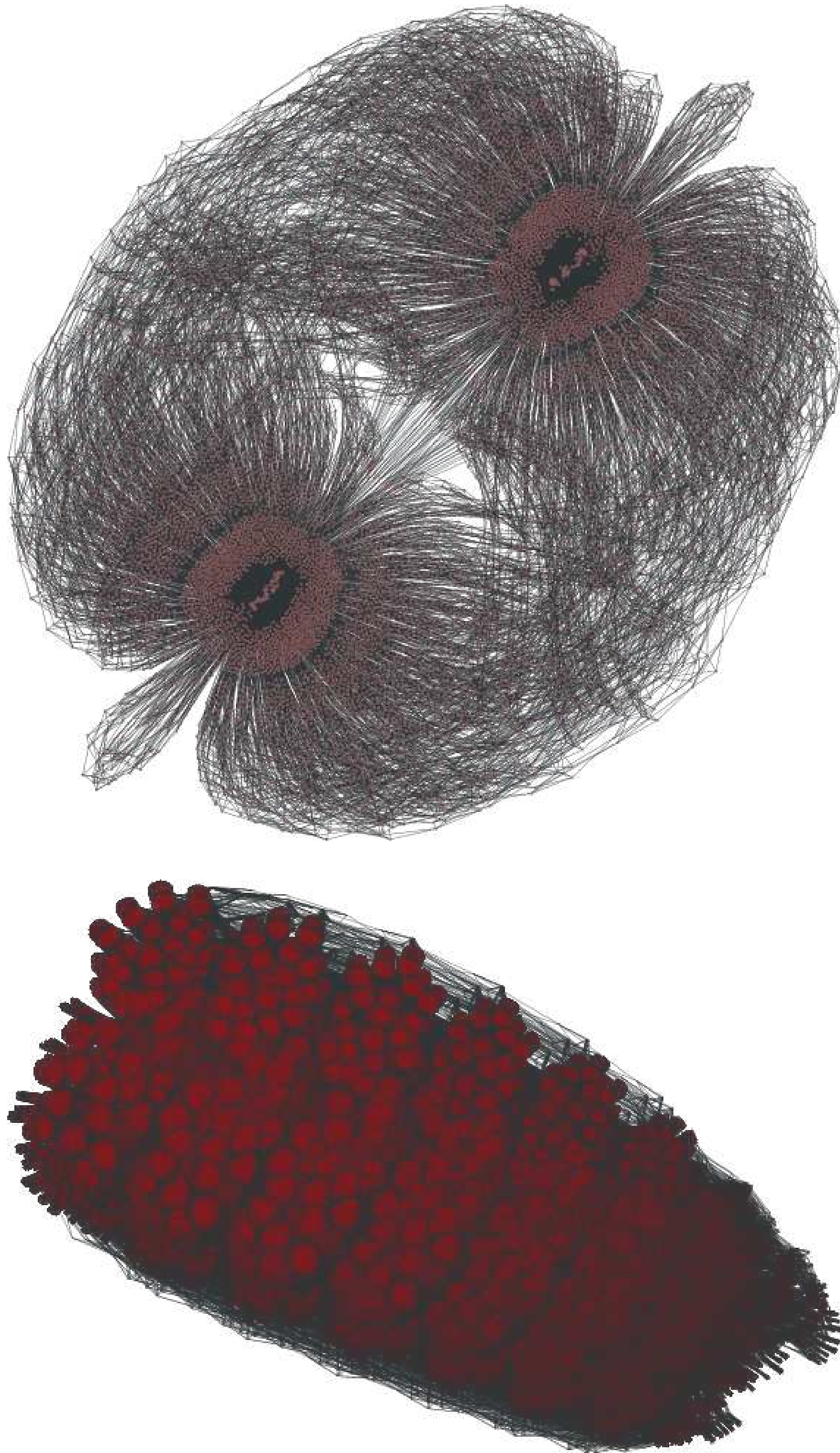
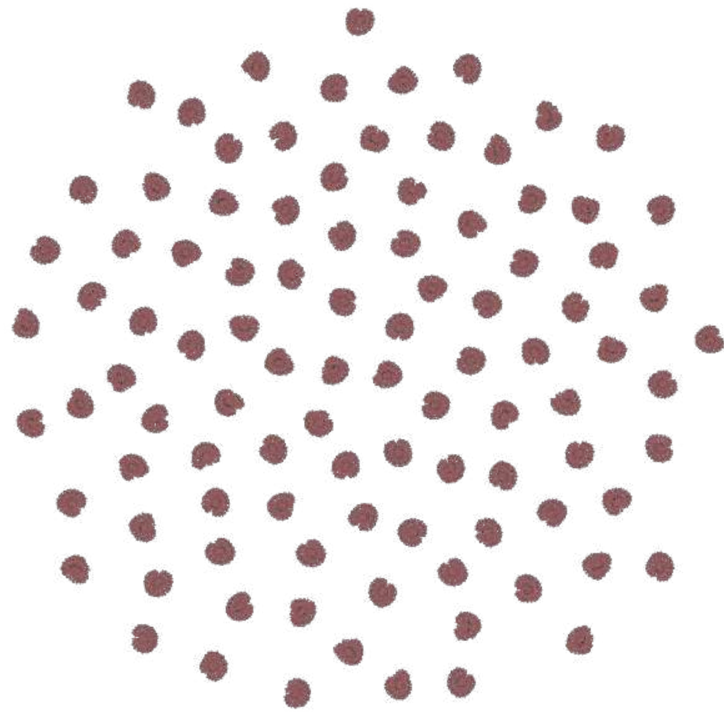


**Figure 3.5:** Variable interaction graphs of preprocessed application instances SAT_instance_N=49 (top) and aaai10-ipc5 (bottom).



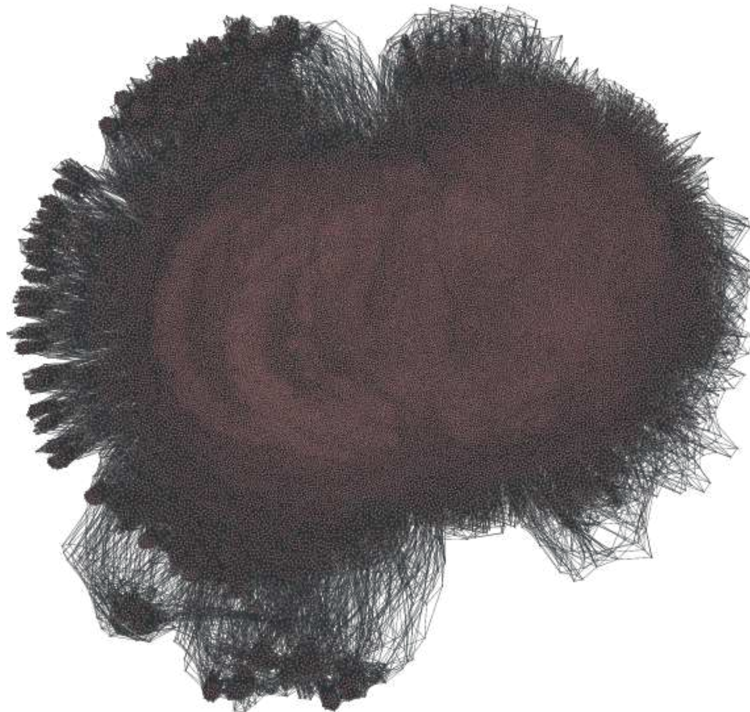


**Figure 3.6:** Variable interaction graphs of preprocessed application instances: prime2209-98 (top) and AProVE09-06 (bottom).

over a hundred independent densely connected components. It appears that such instances might be hard to be decomposed further. Lastly, AProVE09-06 seems uniformly and densely connected at its large core. In this case, it might be difficult to exploit its decomposability, despite its small normalized treewidth.

Through the visual analysis on the VIGs, we find that (normalized) treewidth, while reveals the potential decomposability of an instance, does not always tell the full story. Visualization of VIGs can help gain more intuitive insights, which can sometimes contradict to the intuitions based on treewidth suggests. We will further learn how well can a VIG be decomposed in practice with pseudo backbones in Subsection 3.1.3, and compare the practical decomposability against the theoretical indicators such as the treewidth and the visualization of VIGs.

### 3.1.2 Computing Pseudo Backbone from Good Local Optima

In contrast to the backbone that is extracted from all global optima, pseudo backbones are approximated from local optima with low evaluations (aka "Good Local Optima") [48]. We employ AdaptG$^2$WSAT [16] from the UBCSAT implementation [17] to find good local optima, because both our preliminary experiments and previous study by Kroc *et al*. [152] indicate that AdaptG$^2$WSAT is one of the best performing SLS solvers on application instances. We run AdaptG$^2$WSAT for up to 1000 seconds, record the best local optima found during the run, and repeat the run ten times with different random seeds for each of the 28 preprocessed application instances. As a result, 10 good local optima are collected for each instance.

Table 3.2 presents statistics on how "good" the local optima are. Interestingly, AdaptG$^2$WSAT reliably finds a global optima in all 10 runs within 1000 seconds on prime2209-98. Except for the 5 atco instances, AdaptG$^2$WSAT finds local optima with at most hundreds of unsatisfied clauses, on instances with up to about 3 million clauses. Although these are somewhat good local optima, we also find considerable room for improving upon AdaptG$^2$WSAT , which is one of the best SLS solvers, on application instances that exhibit high potential decomposability.

**Table 3.2:** Quality of the best local optima found by AdaptG$^2$WSAT over 10 of 1000-second runs, on the 28 selected preprocessed application instances with small *tw.n.pre*. It is a minimization problem with 0 being a global optimum, where the evaluation of a candidate solution is the number of unsatisfied clauses. "inst" indicates the instance name. "n.pre" ("m.pre") indicates the number of variables (clauses). "eval.mean" ("eval.std") indicates the mean (standard deviation) evaluations of the best local optima over 10 runs.

| inst | n.pre | m.pre | eval.mean | eval.std |
|---|---|---|---|---|
| aaai10-ipc5 | 308480 | 2910796 | 21.50 | 1.08 |
| AProVE09-06 | 37726 | 192754 | 472.90 | 14.49 |
| atco_enc3_opt1_03_53 | 991419 | 3964576 | 494523.10 | 449.40 |
| atco_enc3_opt1_04_50 | 1320073 | 5313526 | 775484.80 | 922.17 |
| atco_enc3_opt1_13_48 | 1471468 | 5921783 | 906206.90 | 529.18 |
| atco_enc3_opt2_05_21 | 1293612 | 5193145 | 755748.10 | 449.16 |
| atco_enc3_opt2_18_44 | 1067657 | 4305314 | 555417.40 | 906.20 |
| LABS_n044_goal003 | 20259 | 70752 | 4.80 | 1.55 |
| LABS_n045_goal003 | 22095 | 76929 | 6.30 | 1.57 |
| LABS_n051_goal004 | 32522 | 112980 | 5.60 | 1.26 |
| LABS_n052_goal004 | 34674 | 120036 | 6.30 | 1.25 |
| LABS_n066_goal005 | 71336 | 245401 | 10.80 | 1.14 |
| LABS_n078_goal006 | 118601 | 405972 | 11.50 | 1.78 |
| LABS_n081_goal007 | 138803 | 479962 | 75.20 | 7.30 |
| LABS_n082_goal007 | 147213 | 509579 | 167.80 | 10.71 |
| LABS_n088_goal008 | 182015 | 638771 | 654.10 | 67.13 |
| prime2209-98 | 16758 | 88592 | 0.00 | 0.00 |
| SAT_instance_N=49 | 13766 | 54806 | 117.90 | 9.26 |
| SAT_instance_N=55 | 17272 | 68977 | 141.30 | 10.03 |
| SAT_instance_N=63 | 22780 | 90906 | 193.50 | 15.99 |
| SAT_instance_N=69 | 27286 | 109087 | 239.10 | 6.71 |
| SAT_instance_N=72 | 29289 | 117013 | 264.10 | 9.10 |
| SAT_instance_N=75 | 32356 | 129290 | 288.00 | 11.72 |
| SAT_instance_N=77 | 34129 | 136225 | 303.60 | 11.87 |
| SAT_instance_N=85 | 41562 | 166062 | 358.80 | 9.96 |
| SAT_instance_N=93 | 49745 | 199060 | 442.70 | 10.19 |
| SAT_instance_N=99 | 57407 | 229153 | 467.00 | 44.59 |
| SAT_instance_N=111 | 72001 | 287702 | 617.90 | 18.30 |

Here we employ the pseudo backbone concept differently from Zhang's approach [48]. In Zhang's work [48], all local optima are used to compute an empirical probability distribution, *pseudo backbone frequency*, which estimates the likelihood of a variable assigned to true in the true backbone. In our case, however, we instead prefer to maximize the pseudo backbone size for decomposition purpose. We expect that the using more distinct local optima leads to smaller pseudo backbone sizes. We present an analysis of the similarity between the all pairs of local optima as well as the number of fixed variable assignments across all 10 local optima in Table 3.3. From Table 3.3 we observe that, considering the pairs of local optima instead of the 10 local optima all together indeed yields larger sets of fixed variable assignments. We also notice that the variation in the number of fixed variable assignments across all $\frac{10\times(10-1)}{2} = 45$ pairs of local optima is small, as indicated by the small standard deviations ("std.fixed") and the small difference between "min.fixed" and "max.fixed". Meanwhile, the difference between "mean.fixed" and "all.fixed" is much more pronounced. On atco instances, the typical number of fixed variable assignments of pairs of local optima (as indicated by "mean.fixed") is more than $100\times$ larger than that of all 10 local optima (as indicated by "all.fixed"). For the purpose of fixing as many pseudo backbone variables as possible to increase the chance of decomposing a given VIG, we choose the pseudo backbone of the pair of local optima that yields "max.fixed" for simplifying VIGs.

In addition to the finding in Table 3.2 that AdaptG$^2$WSAT consistently finds a optimal global optimum with 1000 seconds in all 10 runs on prime2209-98, Table 3.3 further indicates that AdaptG$^2$WSAT in fact finds the *same* global optimum in all 10 runs. This excludes the instance prime2209-98 (the corresponding row is grayed out in Table 3.3) from future experiments on exploiting decomposability with pseudo backbones, since AdaptG$^2$WSAT can only find one global optimum and hence the pseudo backbone contains all variables, leaving an empty graph after assigning pseudo backbone variables.

**Table 3.3:** Statistics ("min.fixed", "mean.fixed", "std.fixed", and "max.fixed" columns) on numbers of fixed variable assignments in every pair of local optima, and the number of fixed variable assignments across all local optima ("all.fixed" column).

| inst | Between pairs of local optima | | | | all.fixed |
|---|---|---|---|---|---|
| | min.fixed | mean.fixed | std.fixed | max.fixed | |
| aaai10-ipc5 | 271741 | 272964.09 | 585.12 | 274056 | 231715 |
| AProVE09-06 | 20676 | 21407.02 | 335.39 | 22026 | 2669 |
| atco_enc3_opt1_03_53 | 495853 | 497323.18 | 629.96 | 498900 | 2768 |
| atco_enc3_opt1_04_50 | 660175 | 661204.38 | 587.74 | 662712 | 3570 |
| atco_enc3_opt1_13_48 | 736017 | 736955.64 | 574.38 | 738506 | 3910 |
| atco_enc3_opt2_05_21 | 646598 | 648166.36 | 621.79 | 649557 | 4061 |
| atco_enc3_opt2_18_44 | 533617 | 535064.49 | 597.61 | 536835 | 3162 |
| LABS_n044_goal003 | 16626 | 17130.02 | 184.58 | 17661 | 11915 |
| LABS_n045_goal003 | 18141 | 18702.04 | 253.46 | 19412 | 13163 |
| LABS_n051_goal004 | 27368 | 27956.42 | 249.63 | 28465 | 20389 |
| LABS_n052_goal004 | 29194 | 29747.98 | 265.32 | 30397 | 21513 |
| LABS_n066_goal005 | 61959 | 63082.53 | 445.85 | 64135 | 49308 |
| LABS_n078_goal006 | 104272 | 105496.04 | 569.01 | 106950 | 83702 |
| LABS_n081_goal007 | 121057 | 122232.29 | 773.69 | 124133 | 95655 |
| LABS_n082_goal007 | 127530 | 129640.62 | 1466.40 | 132499 | 102097 |
| LABS_n088_goal008 | 134971 | 152445.31 | 7437.70 | 160520 | 103838 |
| prime2209-98 | 16758 | 16758.00 | 0.00 | 16758 | 16758 |
| SAT_instance_N=49 | 7500 | 7971.16 | 209.61 | 8356 | 1017 |
| SAT_instance_N=55 | 9543 | 10189.20 | 332.91 | 10852 | 1513 |
| SAT_instance_N=63 | 12717 | 13285.69 | 308.04 | 14071 | 1875 |
| SAT_instance_N=69 | 15294 | 15824.64 | 332.64 | 16663 | 2008 |
| SAT_instance_N=72 | 16407 | 16984.73 | 336.49 | 17879 | 2196 |
| SAT_instance_N=75 | 17807 | 18750.04 | 417.89 | 19487 | 2224 |
| SAT_instance_N=77 | 18924 | 19685.49 | 410.88 | 20473 | 2300 |
| SAT_instance_N=85 | 22810 | 24243.04 | 551.84 | 25229 | 3232 |
| SAT_instance_N=93 | 27198 | 28693.51 | 705.01 | 30307 | 3050 |
| SAT_instance_N=99 | 33073 | 34138.16 | 610.53 | 35594 | 5534 |
| SAT_instance_N=111 | 41011 | 42401.62 | 683.94 | 44759 | 5986 |

### 3.1.3 Improving Decomposition on SAT Instances

Selecting the pair of local optima that leads to the largest pseudo backbone does not necessarily decompose a given VIG. Two additional confounding factors also come into play. First, the distribution of pseudo backbone variables over the VIG is critical. The pseudo backbone variables might scatter over many tree nodes in the tree decomposition, which impairs the decomposition. Taking Figure 2.2 for an example, assigning variables $B$ and $C$ decomposes the graph into two independent components, whereas assigning variables $C$ and $G$ does not. Second, assigning some variables can lead to considerable applications of unit propagations, which can further simplify the VIG and facilitate the decomposition. In this section, we first introduce two SAT-specific optimizations that lead to better decomposition than on general pseudo Boolean optimization problems in Subsubsection 3.1.3, and then conduct an empirical study in Subsubsection 3.1.3 to evaluate the practical decomposability of VIGs with pseudo backbones we collected in the previous section.

## Theoretical Analysis

In general pseudo Boolean optimization, fixing the assignment to a variable triggers the removal of that variable and all edges incident to it [88]. Due to the specialty of SAT problem, we introduce two optimizations that promotes the decomposition of VIGs, and is not available in general pseudo Boolean optimization.

First, we remove clauses satisfied by the assigned variables. Assigning a pseudo backbone variable $v$ can possibly satisfy a clause $C$, which leads to direct removal of the entire clause $C$ from the formula. Note that each clause forms a clique in the VIG, since every pair of variables in a clause interacts. Suppose $C$ contains $k$ variables and $v$ only appears in $C$, assigning $v$ removes up to $\frac{k\times(k-1)}{2}$ edges (i.e, the number of edges in a clique of size $k$) from the VIG. Now consider a similar pseudo Boolean optimization problem, in which a variable $v$ only appears in a subfunction $C$ that contains $k$ variables. Assigning $v$ in the general pseudo Boolean optimization problem, however, only removes $v$ from $C$, which leads to the deletion of only the $k-1$ edges incident to $v$.

Second, we apply unit propagation after assigning pseudo backbone variables. Unit propagation can also imply the assignments of extra variables besides the pseudo backbone variables. The implied variables again can satisfy some clauses, simplifying and possibly decomposing the VIG even more. In this sense, unit propagation *reinforces* the first optimization. Given the same VIGs, SAT instances clearly have better theoretical potential of being decomposed than general pseudo Boolean optimization instances.

## Empirical Results

Recall that every pair of 10 local optima generates a pseudo backbone. On each application instance, 10 local optima result in $\frac{10\times(10-1)}{2} = 45$ pairs, which are further used to generate 45 pseudo backbones. We then use the pseudo backbones to simplify each of the application instances, which leads to 45 simplified instances for every application instance. For each application instance, the statistics (min, median and max) on the number of connected components of the instance simplified using 45 different pseudo backbones are presented in Table 3.4.

We present the median instead of the mean, because we notice a large variance in the number of connected components on many instances. Taking LABS_n052_goal004 for example, the maximum number of components is $548\times$ the minimum number of components. In fact, we argue that the high variance on the number of connected components is an empirical indicator for partition crossover to be *more powerful on SAT* than on other pseudo Boolean optimization problems like NK-landscapes.

The maximum of number of connected components across all 27 non-empty simplified instances varies from 21 to 2084. By [88], partition crossover can find the best of $2^q$ possible offsprings on $q$ connected components in $O(n)$ time, where $n$ is the number of Boolean variables. This means there is always an opportunity for applying partition crossover. In some cases, one application of partition crossover can return the best among $2^{2084}$ offsprings. As for the typical case scenarios, the median number of connected components varies from 0 to 1373, with an average of 249.

We notice that some instances are drastically simplified. The min and median number of components for two acto instances are zero, meaning that the two simplified instances are com-

**Table 3.4:** Statistics (“min, “median and “max columns) on numbers of connected components after decomposition using pseudo backbones.

| inst | min | median | max |
|---|---|---|---|
| aaai10-ipc5 | 7 | 20 | 38 |
| AProVE09-06 | 11 | 1373 | 1620 |
| atco_enc3_opt1_03_53 | 937 | 1020 | 1090 |
| atco_enc3_opt1_04_50 | 1023 | 1087 | 1164 |
| atco_enc3_opt1_13_48 | 1193 | 1287 | 1365 |
| atco_enc3_opt2_05_21 | 0 | 0 | 37 |
| atco_enc3_opt2_18_44 | 0 | 0 | 21 |
| LABS_n044_goal003 | 1 | 52 | 374 |
| LABS_n045_goal003 | 1 | 52 | 419 |
| LABS_n051_goal004 | 1 | 69 | 544 |
| LABS_n052_goal004 | 1 | 77 | 548 |
| LABS_n066_goal005 | 100 | 133 | 1012 |
| LABS_n078_goal006 | 156 | 179 | 1566 |
| LABS_n081_goal007 | 146 | 291 | 394 |
| LABS_n082_goal007 | 168 | 251 | 1435 |
| LABS_n088_goal008 | 231 | 371 | 2084 |
| SAT_instance_N=111 | 34 | 55 | 1218 |
| SAT_instance_N=49 | 0 | 26 | 156 |
| SAT_instance_N=55 | 0 | 39 | 186 |
| SAT_instance_N=63 | 21 | 32 | 392 |
| SAT_instance_N=69 | 28 | 51 | 519 |
| SAT_instance_N=72 | 12 | 40 | 370 |
| SAT_instance_N=75 | 25 | 41 | 345 |
| SAT_instance_N=77 | 0 | 41 | 719 |
| SAT_instance_N=85 | 27 | 48 | 707 |
| SAT_instance_N=93 | 16 | 46 | 408 |
| SAT_instance_N=99 | 36 | 54 | 1075 |

pletely empty. Note that pseudo backbones on average only consists of roughly 50% of the variables (648166/1293612 = 0.501 for atco_enc3_opt2_05_21 and 535064/1067657 = 0.501 for atco_enc3_opt1_13_48). Therefore, the other half of variables are all implied by unit propagation. This suggests that maximizing pseudo backbone size does not always translate into the maximum number of connected components. There is a *trade-off* between simplifying a graph so that it can be decomposed, and avoiding over-simplifying a graph that leads to fewer or no connected components. Fortunately, the maximum number of connected components for atco_enc3_opt2_05_21 and atco_enc3_opt1_13_48 is 37 and 21, meaning that there are still chances where partition crossover can be applied.

In Figure 3.7, Figure 3.8 and Figure 3.9, we present the decomposed VIGs that yields the median number of connected components. There are several interesting patterns. First, three instances, atco_enc3_opt1_13_48, LABS_n088_goal008 and SAT_instance_N=49, are decomposed into mostly linearly connected components. Second, even though the pseudo backbone for aaai10-ipc5 contains 88% of the variables, the instance still has a large connected component that contains the majority of the remaining variables. Notice that there are several weak links in the largest components that could have been removed using a pseudo backbone, leading to more connected components. This indicates that pseudo backbone indeed can miss some "low-hanging fruit" for further decomposing a VIG, because it is oblivious of the tree decomposition. Third, recall that the VIG of AProVE09-06 before simplification (see Figure 3.6) appears difficult to decompose. Indeed, although the simplified VIG shows that AProVE09-06 is decomposed into many connected components, some of the connected components are non-trivially large and complex, indicating its limited potential of being decomposed further. Visualization can indeed complement treewidth in identifying instances with potential decomposability.

## 3.2 Conclusion

Our work serves as a key stepping stone for applying partition crossover to SAT. In fact, given the promising theoretical principles and the encouraging empirical results, we argue that SAT is

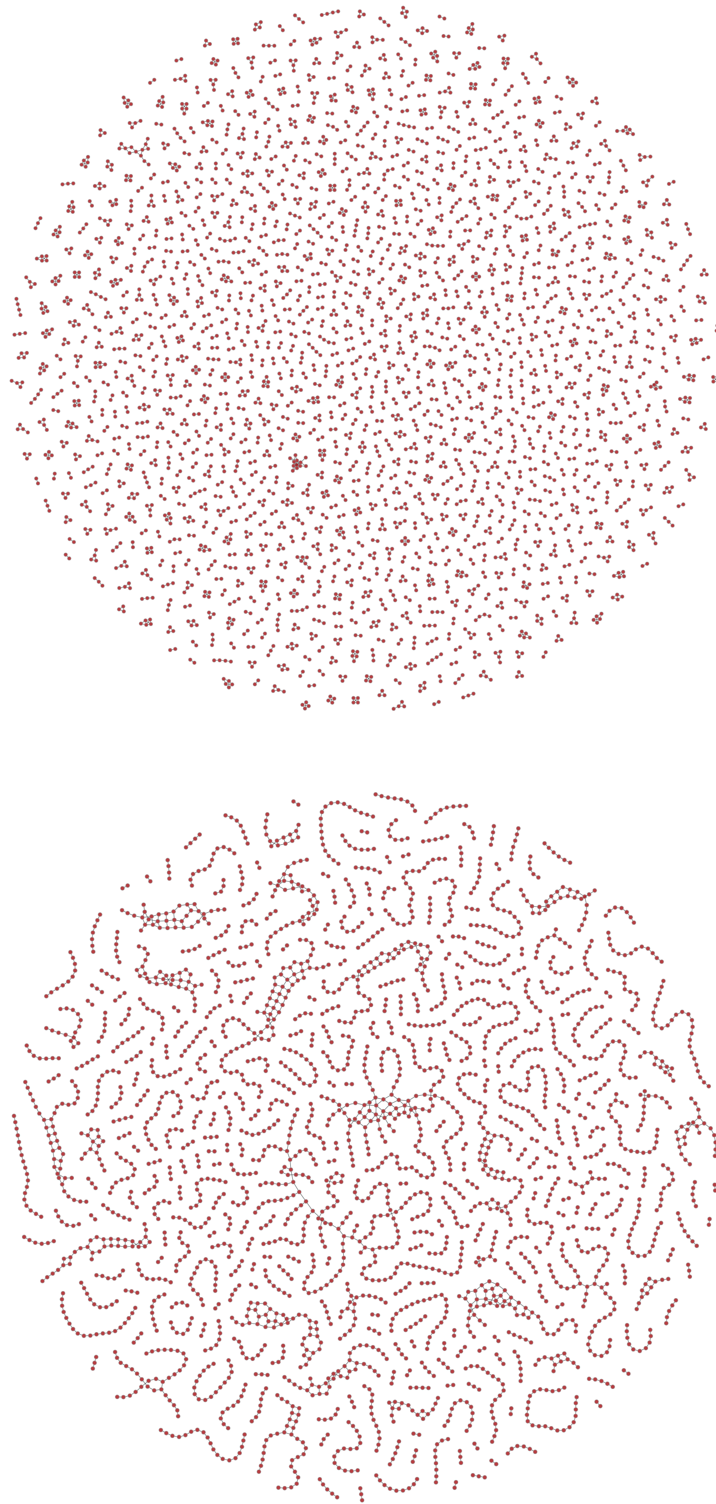
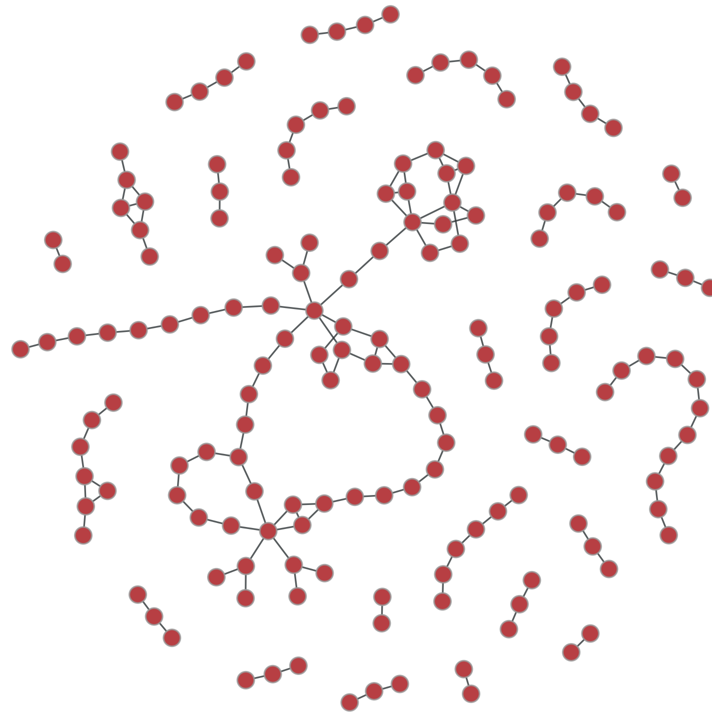


**Figure 3.7:** Decomposed Variable interaction graphs of representative application instances, atco_enc3_opt1_13_48 (top) and LABS_n088_goal008 (bottom), that yields the median number of connected components.



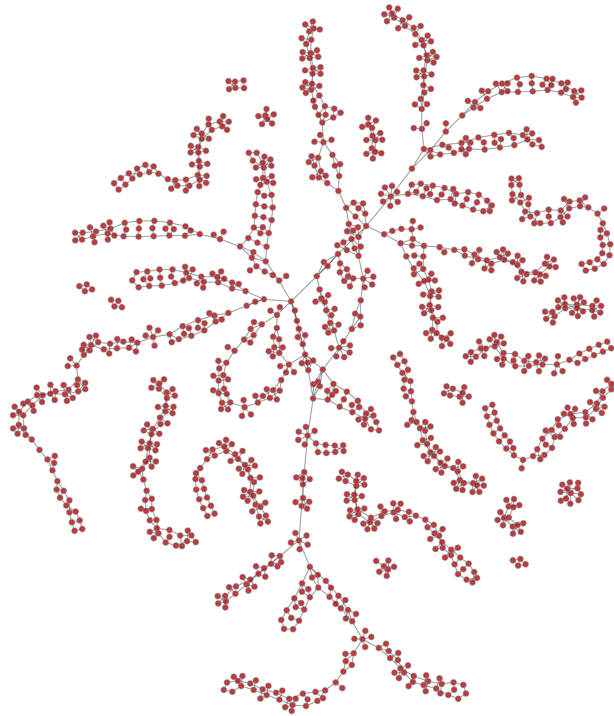


**Figure 3.8:** Decomposed Variable interaction graphs of representative application instances, SAT_instance_N=49 (top) and aaai10-ipc5 (bottom), that yields the median number of connected components.

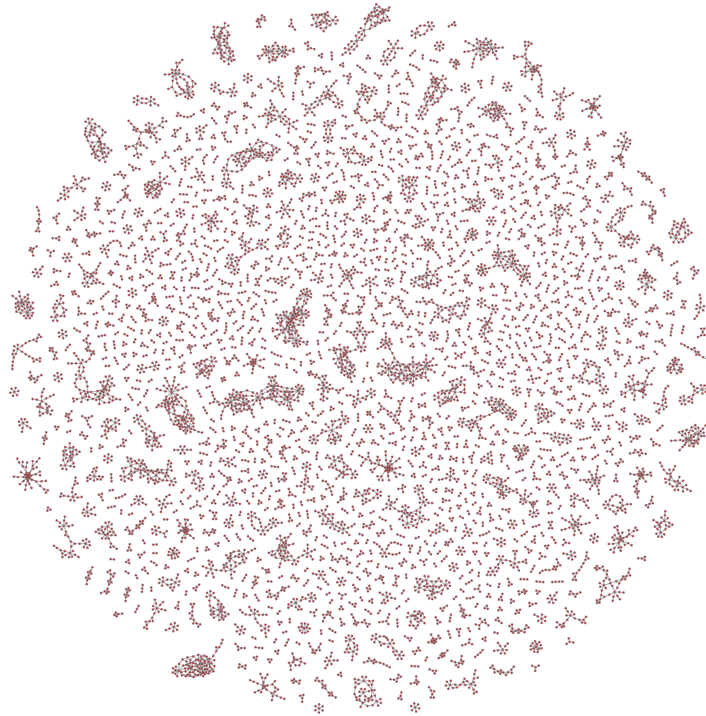


**Figure 3.9:** Decomposed Variable interaction graphs of representative application instance AProVE09-06 that yields the median number of connected components.

even more suitable for applying partition crossover than general pseudo Boolean optimization problems. Our ultimate goal is to narrow the gap in performance between SLS solvers and SS solvers on application SAT instances. Navigating through local optima is known to dominate the running time of SLS solvers [134]. Many local optima are visited during the course of SAT solving with SLS solver. Abandoning the valuable information carried in the local optima seems unwise. Now that we have shown that application instances can be decomposed using pseudo backbone constructed from local optima, applying partition crossover to leverage the numerous local optima while exploiting decomposability of application SAT instances is our proposed work.

# Chapter 4

# Partition Crossover for Improving Stochastic Local Search SAT Solvers [15]

Boolean Satisfiability (SAT) is the first problem proven NP-Complete [2]. Maximum Satisfiability (MAXSAT) is the optimization version of SAT. The goal of MAXSAT is to find an assignment that satisfies the maximal number of clauses. An efficient way of solving many optimization problems, such as automated design debugging [155] and the maximum clique problem [156], is by converting these problems into MAXSAT and applying a MAXSAT solver. MAXSAT-based approaches can even outperform specialized solvers in areas like nonlinear dimensional reduction [157] and Bayesian network learning [158].

The two major search paradigms for solving MAXSAT are Systematic Search such as branch-and-bound solvers [159] and local search algorithms such as the ones in UBCSAT collection [17]. Note that from the perspective of local search, SAT instances are also viewed as optimization problems, since local search finds a satisfiable solution by flipping one bit at each iteration, continuously seeking solutions with fewer unsatisfied clauses. Local search can reliably solve uniform random instances with one million variables and several million clauses to optimality in recent SAT competitions[16]. Despite its demonstrated raw power in solving difficult uniform random instances, local search still suffers from the following two prominent issues.

1. Local search solvers frequently encounter a sequence of states where it is difficult to reduce the number of unsatisfied clauses. Moving through these regions, called *plateau moves*, usually dominates the running time of local search solvers [132, 160]. Furthermore, the valuable history of information accumulated after high quality solutions are visited is typically abandoned, which seems unwise.

[15]This chapter is based on paper [154].

[16]http://satcompetition.org/

2. Local search solvers have poor performance on application SAT instances. Application SAT instances often have internal structure. *Decomposability* focuses on how well the variable interactions of an application instance can be decomposed. Decomposability has been extensively studied and exploited by systematic SAT solvers with success [107, 125]. In contrast, local search solvers completely ignore structure and potential decomposability of application instances.

We present a new framework called *PXSAT*, based on the recombination operator *Partition Crossover (PX)* [88]. The notion of "recombination" (i.e., "crossover") is borrowed from genetic algorithms. However, in this case the operator is deterministic, not stochastic, and PX offers performance guarantees. PX is also designed to be used in combination with local search.

PX takes as input two solutions which are local optima, or otherwise are good solutions found on a plateau of the search space. We can prove that PX is able to create a "tunnel" that directly moves from two known locally optimal solution to arrive at new local optima in $O(n)$ time. It has been used successfully on combinatorial optimization problems such as the Traveling Salesman Problem [161] and NK-Landscapes [162]. It has not previously been applied to MAXSAT. Previous experiments conducted by [48] and [151] provide evidence that high quality local optima typically share partial solutions with optimal solutions. Applying PX to MAXSAT has the potential of finding improving moves that tunnel from one plateau to a better plateau by changing hundreds (or even thousands) of variables at the same time.

PX can also be used to exploit the decomposability of MAXSAT application instances. PX fixes what can be considered pseudo backbone variables (i.e., variable assignments shared among local optima [22]), to locally *decompose* the Variable Interaction Graph (VIG) into $q$ components that are independent from each other. PX then recombines partial solutions from different components such that the best solution among all possible $2^q$ reachable solutions. This occurs in O(n) time. Previous studies report that many application instances do have high decomposability and can be decomposed into up to thousands of components [37, 98]. These results suggest that PX has the potential to be very useful on MAXSAT application instances.

**Table 4.1:** An Example of MAX-3SAT Instance.

| | | | |
|---|---|---|---|
| **a: 1 -3 6** | **l: -6 10 13** | **q: -11 16 17** | **v: -15 -7 -13** |
| **b: 2 -1 6** | **m: 8 -18 6** | **r: 12 -10 17** | **w: 16 -9 -11** |
| **c: -1 2 4** | **n: 7 -12 -15** | **s: -13 -12 15** | **x: 17 -5 -16** |
| **d: -4 1 14** | **o: 9 11 14** | **t: 14 -4 16** | **y: -18 -7 13** |
| **e: -5 4 2** | **p: -10 -2 17** | **u: -9 14 16** | **z: 3 6 -14** |

While applying PX to MAXSAT is in principle simple, doing so while also controlling execution costs is not trivial. State-of-art local search MAXSAT solvers are highly optimized, so that each improving move takes only $O(1)$ time. Each applications of PX takes $O(n)$ time. How to best balance the use of local search and PX on MAXSAT is a nontrivial question. In the current implementation, PX is only triggered when there is no improvement in evaluation in the past $\alpha * n$ iterations.

Another problem in applying PX to MAXSAT is deciding what candidate solutions to recombine. When there are well defined local optima, this is less of a problem. New theoretical finding can guide the design of the PXSAT to avoid triggering unproductive applications of Partition Crossover.

Empirical results on an extensive set of application instances show combining Partition Crossover with the local search algorithms can yield substantially better results. We improve two of best local search solvers, AdaptG$^2$WSAT and Sparrow. PXSAT combined with AdaptG$^2$WSAT is also able to outperform CCLS, winners of several recent MAXSAT competitions.

## 4.1 Variable Interaction and Tunneling

Let $f(x)$ be the evaluation function for an assignment $x \in \mathbb{B}^n$, where $f(x)$ counts the number of unsatisfied clauses. Consider the following MAX-3SAT function composed of the following clauses in Table 4.1.

A clause **a: 1 -3 6** has a label **a** and variables **1 -3 6**. A positive variable (e.g. **1**) is satisfied by an assignment of True. A negated variable (e.g. **-3**) is satisfied by a False assignment. Each clause is in Conjunctive Normal Form; at least one literal must be satisfied for the clause to be satisfied.

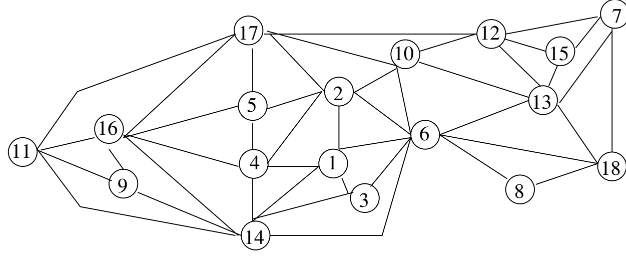



**Figure 4.1:** An illustration of the VIG.

The goal is to maximize the number of satisfied clauses. Let $m$ denote the number of clauses, and $n$ denote the number of variables.

From these clauses, we can extract the nonlinear interactions between the variables. An exact way to compute the nonlinear interactions is to use a discrete Fourier transform to generate a discrete Fourier polynomial; this can be done in $O(n)$ time assuming $m = O(n)$. We can be less exact and assume that if two variable appear together in a single clause, there is a nonlinearity between those variables. The true nonlinear interactions must be a subset of this set. This leads to the following definition:

**Definition 1** (Variable Interaction Graph (VIG))**.** *A variable interaction graph has a set of vertices which are the variables of a MAXSAT instance. If two variables, $x_i$ and $x_j$ appear together in a clause, there is an edge $e_{i,j}$ in the VIG.*

The VIG has at most $3m = O(n)$ edges for MAX-3SAT. Figure 4.1 presents the *VIG* for Table 4.1. Assume we have two candidate solutions $P1$ and $P2$ to Table 4.1 that have been found by local search, where neither solution can be improved by a single bit flip.

$$P1 = 00000\ 00000\ 00000\ 000$$

$$P2 = 11100\ 01110\ 11101\ 101$$

$P1$ satisfies all of the clauses except clause **o** but flipping bits 9 or 11 or 14 causes clauses **q** or **u** or **z** respectively to be unsatisfied. $P2$ satisfies all of the clauses except clause **v** but flipping bits 15 or 13 or 7 causes clauses **s** or **y** or **m** to be unsatisfied.

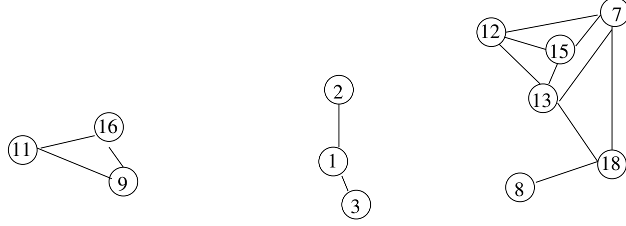



**Figure 4.2:** The Recombination Graph with three separable recombining components for the parent $P1 = 00000\ 00000\ 00000\ 000$ and $P2 = 11100\ 01110\ 11101\ 101$.

Clearly, $x_4 = x_5 = x_6 = x_{10} = x_{14} = x_{17} = 0$ in $S_{P1}$ and $P2$. For all other bits, $x_i = 0$ in $P1$, and $x_i = 1$ in $P2$. The two solutions are contained in the hyperplane ***000***0***0**0* where $*$ denotes the bits that are different in the two solutions, and $0$ marks the positions where variables share the same assignment.

We use the hyperplane ***000***0***0**0* to decompose the VIG to generate a *recombination graph*. We remove all of the variables (vertices) that have the same assignments and remove the edges incident on the removed vertices. The *Recombination Graph* is shown in Figure 4.2.

The recombination graph breaks the VIG into connected subgraphs, which we will define as *recombining components*. In Figure 4.2 there are $q = 3$ recombining components. Variables that are connected in the recombination graph represent complementary partial solutions. The recombination graph also decomposes the evaluation function $f(x)$ into linearly separable subfunctions. Thus, in Table 4.1 we can define a new subfunction $g(x')$ such that

$$g(x') = a + g_1(x_9, x_{11}, x_{16}) + g_2(x_1, x_2, x_3) \\ + g_3(x_7, x_8, x_{12}, x_{13}, x_{15}, x_{18})$$

where $a$ is a constant and $g(x') = f(x)$ but where the domain of function $g(x')$ is restricted to the largest hyperplane subspace containing strings $P1$ and $P2$. Since clause **o** unsatisfied by $P1$ and clause **v** unsatisfied by $P2$ are in different components of recombination graph in Figure 4.2, the offspring (e.g., 00000 00010 10000 100) is guaranteed to improve upon both parents. In this case, PX instantly jumps to the global optimum with all clauses satisfied.

In general the recombination graph induces a new evaluation function

$$g(x') = a + \sum_{i=1}^{q} g_i(x') = f(x)$$

where each subfunction $g_i(x')$ evaluates one connected recombining component of the recombination graph. Let $G_i$ denote the recombining component corresponding to subfunction $g_i(x')$. Note that if there are 2 parents and $q$ recombining components, these partially solutions can be recombined in $2^q$ ways; we refer to this as the set of *reachable solutions*. We can now prove the following result:

**Theorem 1** (PX Theorem [88])**.** *Given a recombination graph with $q$ recombining components, Partition Crossover (PX) returns the best of $2^q$ reachable solutions in O(n) time.*

*Proof.* Because the function $g(x')$ is linearly separable, we can greedily select the best partial solution from $P1$ and $P2$ independently for each subfunction $g_i$. The $q$ greedy choices yields the best of $2^q$ reachable solutions. □

Of course, an improvement only occurs if $P1$ and $P2$ have *recombining components* that also have different evaluations, even when $f(P1) = f(P2)$.

### 4.1.1 Tunneling between Local Optima

Tunneling methods in the form of PX are able to take two solutions as input that are locally optimal, and return a new solution that is also a local optimum in $g(x')$. Thus, tunneling is able to do something that local search cannot: move directly from known local optima to new local optima in one step.

Tunneling methods have already been developed for the Traveling Salesman Problem (TSP) and are a critical part of the Lin Kernigham Helsgaun (LKH) algorithm [163], the best iterated local search algorithm for the TSP. Tunneling methods have also been developed for general k-bounded pseudo-Boolean optimization problems.

Because MAX-kSAT problems are characterized by numerous large plateaus, it is more difficult to characterize the "tunneling" behavior of PX on MAX-kSAT. But for Weighted MAX-kSAT (and k-bounded pseudo-Boolean optimization problems in general) we can prove the following result.

**Theorem 2** ( [88]). *Assume that solutions $P1$ and $P2$ are well defined local optima on a Weighted MAX-kSAT instance. All of the $2^q$ reachable solutions are also local optima under function $g(x')$.*

*Proof.* Assume bit $x_b$ is referenced by both $f(x)$ and $g(x')$. Because $g(x')$ is linearly separable $x_b$ appears in only one subfunction $g_i(x')$. Flipping bit $x_b$ in $f(x)$ will not yield an improving move since $P1$ and $P2$ are local optima; thus, flipping bit $x_b$ in $g_i(x')$ also cannot yield an improving move. □

To be clear, a solution that is locally optimal in $g(x')$ might not be locally optimal in $f(x)$, but if a solution is not locally optimal in $f(x)$ the improving move can only result from flipping a bit assignment shared in common by $P1$ and $P2$. We have examined hundreds of k-bounded pseudo-Boolean functions, and empirically we find that the best of the $2^q$ reachable solutions is also a local optimum in $f(x)$ more than 80 percent of the time.

Since the proof of Theorem 2 is expressed in terms of improving moves, we also automatically get the following result for free. Theorem 3 implies that PX is efficient in the sense that it is difficult to further improve any of the $2^q$ reachable solutions using local search.

**Theorem 3** ( [88]). *Assume that solutions $P1$ and $P2$ are on plateaus such that there is no improving move from solution $P1$ or from $P2$. There are no improving moves from any of the $2^q$ reachable solutions under the function $g(x')$.*

This implies that Partition Crossover is efficient in the sense that it is difficult to further improve any of the $2^q$ reachable solutions using local search.

### 4.1.2 The Cost of PX compared to Local Search

Modern MAXSAT local search algorithms are able to find Hamming distance 1 improving moves in $O(1)$ time using techniques like gradient-based promising variable selection [164]. This

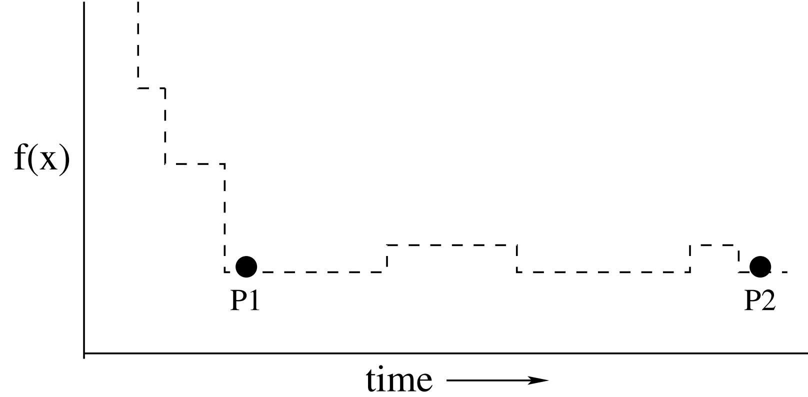



**Figure 4.3:** This figure illustrates how PX is combined with local search. The dashed line tracks changes in $f(x)$. When a plateau is reached, a solution $P1$ is captured. After $\alpha n$ moves with no improving moves, another solution $P2$ is selected, and PX is used to recombine $P1$ and $P2$.

means that PX should only be applied when local search has difficulty finding an improving move, which typically means that search has become stuck on a plateau. PX has the potential to find tunneling moves to an improved solution on a better plateau.

A illustration of how PX is combined with local search appears in Figure 4.3. When a new plateau is reach, a solution $P1$ is recorded. If an improving move is not found after $\alpha n$ moves, another solution on the plateau, $P2$ is selected. Note there are $\alpha n$ moves separating $P1$ and $P2$, which also implies that local search cannot easily escape from the plateau. $P1$ is then recombined with $P2$.

We can efficiently evaluate recombining components of the recombination graph using the $Score$ vector used by all modern local search algorithms to track improving moves (i.e., “makes" and “breaks"). Assume there are candidate solutions $P1$ and $P2$ which decomposes the VIG into $q$ partitions. Assume the current $Score$ vector is defined related to the current solution $P2$. Note that the solution $P1$ and $P2$ have complementary assignments for each subfunction $g_i(x')$. To evaluate the recombining component $G_i$ ($i \in [1, q]$), flip all of the bits in $G_i$, updating the $Score$ vector after each bit flip. The sum of the changes provided by the Score vector is equal to the change in evaluation between $g_i(P2)$ and $g_i(P1)$. While this reduces the runtime, this cost of PX is still $O(n)$.

The following theorem addresses the trade-off in applying PX or doing $\theta(n)$ steps of local search.

**Theorem 4.** *Given $\theta(n)$ time, where $n$ is the number of variables, local search with gradient-based variable selection checks $\theta(n^2)$ candidate solution, while PX checks $\theta(2^q)$ candidate solutions, where $q$ is the number of components in the recombination graphs.*

*Proof.* In local search with gradient-based promising variable selection, an improving move among $n$ neighbor solutions can be discovered in $\theta(1)$ time. Given $\theta(n)$ time, $\theta(n)$ such moves can be performed. A total of $\theta(n^2)$ candidate solutions can thus be checked given $\theta(n)$ time. For PX, only $\theta(1)$ crossovers can be performed in $\theta(n)$ time, so that a total of $\theta(2^q)$ candidate solutions are checked. □

### 4.1.3 Equal Move for PX on Plateau

The search space for MAXSAT is known to contains many neighboring states that share the same evaluation [134]. Such state is called *plateau*. Since local search has no gradient to follow on plateaus, taking equal moves that no change in evaluation certain chance for escaping plateaus. Prior studies [165, 166] have shown that accepting equal moves improve the performance of local search. Plateau can be defined similarly for PX.

**Definition 2** (Component Score)**.** *Given two candidate solutions $P1$ and $P2$, where $P1, P2 \in \mathbb{B}^n$, the pseudo backbone formed by $P1$ and $P2$ decomposes the VIG into $q$ partitions. For a given component $G_i$ ($i \in [1, q]$), the contribution of variable assignments from $P1$ on $G_i$ is denoted as $g^i(P1)$. Component Score, denoted as $Score(G_i)$, going from $P1$ to $P2$ on $G_i$ is defined as*

$$Score(G_i) = g_i(P1) - g_i(P2) \tag{4.1}$$

Similar to the definition of the $Score$ vector in local search [14], component score is the change in $G_i(x)$ when moving from $P1$ to $P2$. Assuming minimization, if the component score is positive, the assignment to $G_i$ in $P2$ is selected. Otherwise, the assignment to $G_i$ in $P1$ is selected.

**Definition 3** (PX Plateau)**.** *Given two candidate solutions $P1$ and $P2$ with $f(P1) = f(P2)$, $P1$ and $P2$ are on a PX plateau if $\forall i \in [1, q], Score(G_i) = 0$.*

**Algorithm 1** Equal Move for Plateau PX

1: **while** $P1$ and $P2$ on PX plateau **and** equal move $i$ exist **and** $P1[i] = P2[i]$ **do**
2: $\quad P2[i] \leftarrow 1 - P2[i]$

Intuitively, PX plateau means taking partial solutions from either parents makes no different in evaluation function, therefore recombination cannot yield an improving offspring. Now consider an equal move mechanism for escaping PX plateaus in 1. The idea is to keep flipping one common bit, so that the two parents become more distant in the search space while remaining equal in evaluation, until it yields an improving offspring by using PX or equal moves are exhausted. While Algorithm 1 provides a seemingly good strategy for escaping PX plateau by increasing the number of components in the recombination graph, Theorem 5 shows it can never yield an improving offspring. Nevertheless, this new theoretical finding can be used to avoid unproductive application of PX.

**Theorem 5.** *Using Algorithm 1, two candidate solutions $P1$ and $P2$ that are on a PX plateau can never escape the plateau.*

*Proof.* Flipping a common variable adds one vertex $v$ back to the recombination graph. Two situations are possible. First, $v$ is isolated from the existing components. The previous components are unaffected by $v$, and remain the zero in component score. $v$ becomes an isolated component $G^*$ with a single variable. Since flipping $v$ doesn't change the evaluation, the score of $G^*$ is also zero. All components remain zero in score. Second, $v$ joins one of existing components $G_i$. Flipping all variables in $G_i$ is known to not change the evaluation by the definition of PX Plateau. Provided that flipping $v$ also does not change the evaluation, flipping variables in the augmented component $G_i \cup \{v\}$ does not change the evaluation. □

## 4.2 PXSAT

In light of the theoretical results, we discuss how they are used to develop guidelines for PX application, which drives the design of our successful PX-based framework for MAXSAT. By

uncovering the design process, we hope to inspire future refinements to PX for MAXSAT as well as new applications of PX to more problem domains.

**Choice of Parents**. According to Theorem 4, $q$ needs to be as large as possible in order to achieve maximal efficiency of PX. For a given instance, $q$ depends on the choices of the two parents $P1$ and $P2$. The hamming distance between $P1$ and $P2$, denoted as $\mathcal{D}(P1, P2)$, determines the number of vertices in the recombination graph, since variables that share assignments are removed. Consider two extreme cases. One is when $\mathcal{D}(P1, P2) \rightsquigarrow 0$, $P1$ and $P2$ have almost identical assignments, there are few vertices left in the recombination graph. The nearly empty recombination graph contains few independent components. The other extreme case is when $\mathcal{D}(P1, P2) \rightsquigarrow n$, $P1$ and $P2$ have little in common. Few vertices have been removed in recombination graph, which stays mostly connected. In this case, PX loses its purpose of decomposing VIG using pseudo backbone.

**Rule 1.** $\mathcal{D}(P1, P2)$ *should neither be too small nor too large.*

Another important guideline on the choice of parents is that $P1$ and $P2$ need to be close in evaluation for a better chance of yielding an improving offspring. Consider the situation where $f(P1) \ll f(P2)$. Partial solutions from $P1$ probably dominate $P2$ in all components. Denote $px(P1, P2)$ as the resulting offspring after applying PX to two parents $P1$ and $P2$. In such cases, $px(P1, P2)$ selects partial solutions exclusively from $P1$. Thus, $px(P1, P2) = P1$. No improving offspring can be discovered via PX.

**Rule 2.** $f(P1) \approx f(P2)$.

Finally, discovering an improving offspring that is better than both parents $P1$ and $P2$ in terms of evaluation is not sufficient. More importantly, $px(P1, P2)$ needs to improve upon the best-so-far solution *bsf*. This requires $P1$ and $P2$ to already contain partial solutions of good quality. The global evaluation of $P1$ and $P2$ should be close to $bsf$.

**Rule 3.** $f(P1) \approx f(bsf)$ *and* $f(P2) \approx f(bsf)$.

**When to Apply PX**. The purpose of this paper is to demonstrate the utility of PX in *complementing* local search in escaping plateaus. We are not replacing local search completely with PX. Instead, we

**Algorithm 2** PXSAT: A Generic Framework based on PX

```
1:  x ← rand();                                   ▷ random initialization
2:  x_best ← x; e_best ← f(x); i ← 0; i_best ← 0;
3:  while termination condition not met do
4:      x ← LS(x);                                ▷ one bit flip by local search
5:      if f(x) < e_best then                     ▷ improvement
6:          x_best ← x; e_best ← f(x); i_best ← i;
7:      else if i > i_best + αn then              ▷ stagnation
8:          x ← px(x, x_best);
9:          ▷ reset regardless of outcome of PX
10:         x_best ← x; e_best ← f(x); i_best ← i;
11:     i ← i + 1
```

present a generic framework that incorporates PX on top of local search. We design the framework with simplicity in mind, hoping that the framework can be incorporated into any existing local search solvers with ease. With the introduction of PX, some of computational resources previously spent on local search inevitably needs to be allocated for PX. Inspired by Theorem 4, we cautiously apply PX when there is no improvement in evaluation for the past $\alpha * n$ iterations. Doing this has two benefits. First, the cost of one application of PX can be amortized over $\alpha * n$ iterations, resulting in an amortized cost per iteration for PX. Second, since local search operators is unable to find a new better solution in the last $\alpha * n$ iterations, introducing PX offers new opportunities of exploring a very different set of candidate solutions on plateaus that are deem difficult to escape by local search.

We now present PXSAT framework in Algorithm 2. $x_{best}$ keeps tracks of the best candidate solution the best solution within a variable-length interval. At initialization, randomly generate the current solution $x$. $x_{best}$ is set to $x$. When local search improves and updates $x_{best}$ in less than $\alpha * n$ steps, the interval keeps expanding. Otherwise, when there is no improvement over $x_{best}$ for $\alpha * n$ steps, a stagnation is detected. In case of stagnation, apply PX to $x_{best}$ and $x$, reset $x_{best}$ to $x$ which start a new interval.

In PXSAT, $\alpha$ is a constant that can be used to indirectly control $\mathcal{D}(x, x_{best})$ such that Rule 1 is conformed. As revealed later in the empirical results section, the optimal setting for $\alpha$ varies by problem instances. Notice that PXSAT applies PX to $x$ and $x_{best}$ rather than the true best-so-far

solution *bsf*. It is because the evaluation of $bsf$ is much better than $x$ in most cases, which breaks Rule 2. $x$ and $x_{best}$, however, are no more than $\alpha * n$ iterations away and are expected to be close in evaluation. Furthermore, using $x_{best}$ to approximate *bsf* also conforms to Rule 3.

## 4.3 Empirical Results

PXSAT is first used to improve two of best performing local search SAT solvers on application instances, AdaptG$^2$WSAT [16] and Sparrow [40]. This shows *relative performance improvement* achieved by incorporating PXSAT into existing solvers. We then demonstrate that the improvement achieved by PXSAT is remarkable by comparing AdaptG$^2$WSAT-PX and Sparrow-PX with CCLS [167], a state-of-art local search solver designed specifically for MAXSAT. This exhibits the *absolute performance* of PXSAT. Empirical results show that the performance of a 10-year-old SAT solver AdaptG$^2$WSAT can be lifted by PXSAT so that the new algorithm even outperforms the state-of-art MAXSAT solver on every instance tested. Finally, we establish a theoretical model for predicting and understanding the success of PXSAT.

### 4.3.1 Setup

To select a wide variety of application instances without biasing any particular class of instances, the benchmark set is constructed as follows. From 150 satisfiable instances from the *crafted track* and 150 satisfiable instances from the *industrial track* in the SAT competition 2014, sample three instances (smallest/median/largest in size in terms of number of variables) from each class if there are more than three, otherwise select all instances. We only selected the satisfiable instances so that the optimum is known to have a zero evaluation. This yields 102 instances, excluding nine instances whose VIGs are too large to process under the 8GB memory limit. All 102 selected instances are preprocessed offline by SatELite [102]. We also evaluated PXSAT on crafted instances and industrial instances from MAXSAT Evaluation 2016[17]. However, these instances were not

[17] http://maxsat.ia.udl.cat

challenging enough; local search is usually able to find an improving solution within $\alpha * n$ iterations. Therefore, PX is rarely triggered.

Preliminary experiments indicate that the parameter $\alpha$ varies by instance classes. As the benchmark set covers a variety of instances from different classes, the parameter $\alpha$ in PXSAT is fixed to be 1, 2, 4, 8 and 16 on each instance, and the best performance among the five settings are used for comparison with the original SAT solvers. The average of best solutions found over 10 trials are first compared. Moreover, we have also learned from our studies that there is a bell-shape relationship between $\alpha$ and the number of components. Finding the optimal setting of $\alpha$ can be achieved by performing a golden section search that successively narrows the range containing the optimal $\alpha$.

### 4.3.2 Improving State-of-Art Local Search SAT Solvers

PXSAT is implemented into AdaptG$^2$WSAT [16] and Sparrow [40]. AdaptG$^2$WSAT has been found to be one of the best performing local search solvers on application instances [152]. Sparrow, on the other hand, performs the best among all local search solvers in both crafted SAT track and application SAT track in SAT Competition 2014[18]. Both solvers are available from the UBCSAT website [19] [17]. UBCSAT provides an efficient implementation of many local search solvers with a unified interface, allowing straightforward incorporation of PXSAT.

Table 4.2 summarizes the impact of PXSAT when incorporated into AdaptG$^2$WSAT and Sparrow. There are 13 instances for AdaptG$^2$WSAT and 11 instances for Sparrow that are easy enough to solve without any stagnation of over $\alpha n$ iterations. On the remaining instances, PXSAT has a *strong positive influence* on both average solution quality and average solving time. Mann-Whitney test [168] is employed to test whether the differences in averages are statistically significant, assuming a significance level of 0.05. We use *percentage improvement* to measure the improvements. It is defined as

---

[18]http://www.satcompetition.org/2014/

[19]https://github.com/dtompkins/ubcsat/releases/tag/v1.2beta18

**Table 4.2:** Comparing PXSAT versions with the original local search solvers. "#No Diff (No PX)": number of instances where PX is never triggered. "#No Diff (PX Ties)": number of instances where PX is triggered and average solving time and average solution quality are tied between the original solver and its PXSAT version. "# Better (Worse) $\Delta sol$": number of instances where the PXSAT version has better (worse) average solution quality. "# Faster (Slower) $\Delta time$": number of instances where the PXSAT version has faster (slower) average solving time.

| Base Solver | AdaptG$^2$WSAT | Sparrow |
|---|---|---|
| #No Diff (No PX) | 13 | 11 |
| #No Diff (PX Ties) | 4 | 2 |
| # Better $\Delta sol$ (Total) | 63 | 73 |
| # Better $\Delta sol$ (Sig) | 38 | 40 |
| # Worse $\Delta sol$ (Total) | 8 | 6 |
| # Worse $\Delta sol$ (Sig) | 2 | 0 |
| # Faster $\Delta time$ (Total) | 11 | 8 |
| # Slower $\Delta time$ (Total) | 3 | 2 |

$$\Delta = (orig - pxsat)/orig \times 100\%, \tag{4.2}$$

where $orig$ is the metric ($\Delta sol$ denotes solution quality and $\Delta time$ denotes solving time) for the base solver and $pxsat$ is for its PXSAT version.

PXSAT improves AdaptG$^2$WSAT on 63 instances in terms of average solution quality; it is worse on only 8 instances. The percentage improvement in average solution quality is 22.4%. The improvements in solution quality achieved by PXSAT on 38 instances are statistically significant. AdaptG$^2$WSAT-PX is only significantly worse on two instances. Considering only the instances where the differences are statistically significant, the percentage improvements in solution quality boosts to 27.2%. This is a substantial improvement considering how difficult it is for AdaptG$^2$WSAT to find a better solution. On average, AdaptG$^2$WSAT stagnates through the last 54.9% of total solving time and local search is unproductive. On the instance `5-SATISFIABLE`, AdaptG$^2$WSAT fails to find any improving solution in the last 4965 (= 99.3% $\times$ 5000) seconds. PXSAT also accelerates AdaptG$^2$WSAT on 11 instances, and slows down AdaptG$^2$WSAT on 3 instances, with an average time reduction of 25.6%. However, due to the stochastic nature of local search where fluctuation in solving time is high, there are only a couple of instances whose solving time differences are statistically significant.

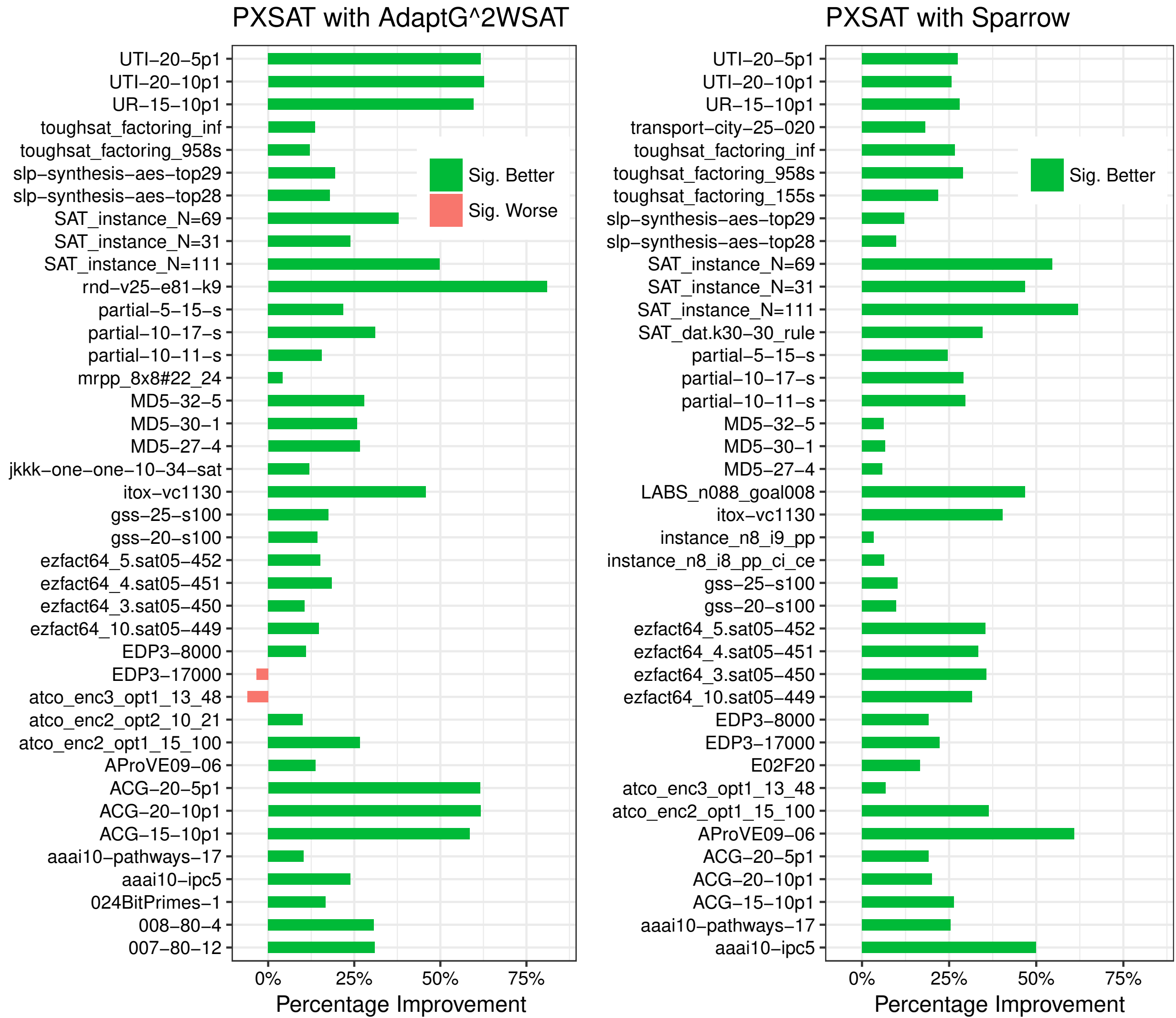



**Figure 4.4:** Comparing PXSAT versions with the original local search solvers on instances where the average solution quality differences are statistically significant.

For Sparrow, PXSAT improves on 73 instances in terms of average solution quality, out of which 40 are statistically significant. There is not a single instance where adding PXSAT results in poorer average solution quality. The average percentage improvement in solution quality over all instances with statistically significant differences is 26.4%. Despite an average improvement of 25.1% on 10 instances where the solving times are different, none of the difference is statistically significant.

Interestingly, for both AdaptG$^2$WSAT and Sparrow, there are 40 instances where the differences in solution quality are statistically significant. Are the set of 40 instances the same for AdaptG$^2$WSAT and Sparrow? Figure 4.4 details the differences on the theses instances. When

PXSAT makes a significant impact on the solution quality, it is almost always a positive impact across the board, except on only two instances. Moreover, the decrease in solution is rather small (3% and 5% respectively), compared with the gain in all the other instances. There are 33 overlapping instances between the two sets. This suggests that PXSAT is particularly useful for a specific set of instances. We next study the common properties they share and how the properties benefit PXSAT.

### 4.3.3 Why and When PXSAT works?

Using Theorem 4, we attempt to understand why PXSAT can improve the solution quality on many instances. The benefit of applying PX $\tau$ times can be quantified in terms of number of inspected candidate solutions as

$$r = \frac{\sum_{i=1}^{\tau} 2^{q_i}}{\tau \times n^2}, \tag{4.3}$$

where $\sum_{i=1}^{\tau} 2^{q_i}$ can be prohibitively large. Instead, we record the average number of component $\overline{p} = \sum_{i=1}^{\tau} q_i/\tau$, and prove that $\overline{p}$ can be used to derive a lower bound of $r$.

**Theorem 6.** *Let* $\check{r} = \frac{2^{\overline{p}}}{n^2}$*, then* $\check{r} \leq r$.

*Proof.* $\check{r} \leq r$ is equivalent to

$$2^{\frac{1}{\tau}\sum_{i=1}^{\tau} q_i} \leq \frac{1}{\tau}\sum_{i=1}^{\tau} 2^{q_i}. \tag{4.4}$$

Jensen's inequality [169] states, if $X$ is a random variable and $\varphi$ is a convex function,

$$\varphi\left(\mathbb{E}[X]\right) \leq \mathbb{E}\left[\varphi(X)\right] \tag{4.5}$$

where the equality holds if and only if $x_1 = x_2 = ... = x_n$ or $\varphi$ is linear. Let $\varphi(q) = 2^q$. Since exponential function is a convex function, we have

$$2^{\frac{1}{\tau}\sum_{i=1}^{\tau} q_i} = \varphi\left(\mathbb{E}[q]\right) \leq \mathbb{E}\left[\varphi(q)\right] = \frac{1}{\tau}\sum_{i=1}^{\tau} 2^{q_i}. \tag{4.6}$$

□

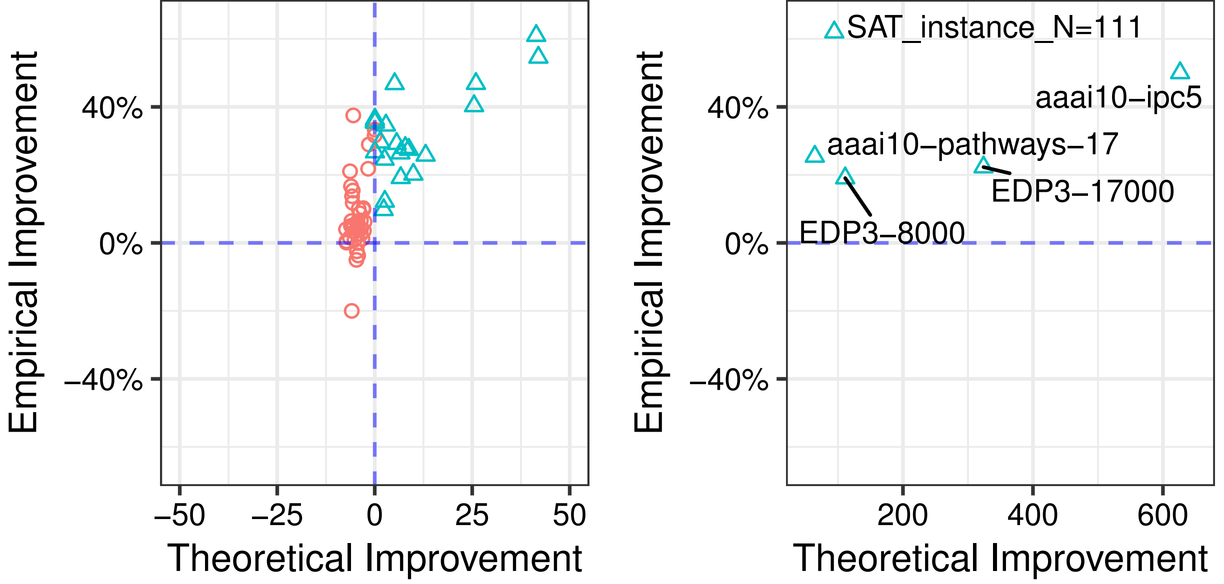



**Figure 4.5:** Empirical $\overline{\Delta sol}$ versus theoretically approximated $\check{lr}$ for Sparrow. Left subfigure has a scale of [-50,50] on X-axis, while right subfigure has a scale of [50,650] on X-axis. Points with $\overline{\Delta sol} > 0$ ($\overline{\Delta sol} < 0$) are colored **blue** (**red**).

In practice, we find that $\check{r}$ is still too large to process, as $\overline{p}$ can be over 1000. We apply log10 transformation to $\check{r}$, and define the result as $\check{lr}$.

$$\check{lr} = log10(\check{r}) = \overline{p} \times log10(2) - log10(n^2). \tag{4.7}$$

$\check{lr}$ is now manageable for processing.

Based on the empirical results and the number of components collected for Sparrow, we perform a correlation analysis between $\overline{\Delta sol}$ and $\check{lr}$. Figure 4.5 presents the outcome. The figure is split into two subfigures with different scales on X-axis, because $\check{lr}$ on some instances, whose names are labeled in the right subfigure, are exceptionally large. This is actually a good news. Take aaai10-ipc5, the one with largest $\check{lr}$ (= 626) for example. This means, given same amount of time, the number of candidate solution inspected is *at least* $\check{r} = 10^{626}$ *times* more than the greedy operators in modern local search. This is a result of the instance decomposed into as many as *over 2117 components* in one application of PX.

**Table 4.3:** Summary statistics on number of components ($q$) and PX Success Rate at finding improving moves for Sparrow-PX.

| | Min | Median | Mean | Max |
|---|---|---|---|---|
| $q$ | 2.000 | 16.707 | 129.482 | 2897.361 |
| Success Rate | 0.01% | 50.2% | 43.2% | 96.6% |

There is a clear positive correlation between $\overline{\Delta sol}$ and $\check{lr}$, which is confirmed by Spearman correlation of 0.656[20]. Given an instance, increasing the number of component is indeed critical for the performance of PXSAT. On the 27 instances where $\check{lr} > 0$, PXSAT always improves $\overline{\Delta sol}$.

Interestingly, there are 35 instances where $\check{lr} < 0$ (which suggests PXSAT inspects less candidate solutions) and yet $\overline{\Delta sol} > 0$ (empirical results show PXSAT improves the performance). Two (non-exclusive) reasons exist. First, $q_i$ across multiple applications of PX is probably very different, thus $\check{r}$ underestimates $r$, which results into a smaller $\check{lr}$. Second, PXSAT is exploring a neighborhood that is drastically different from the one-bit neighborhood in local search solvers. Using PXSAT is beneficial because PX can find improving moves when local search cannot, even if PXSAT checks fewer candidate solutions.

Table 4.3 presents summary statistics on number of components and success rate of PX applications. Despite small median number of components ($\approx 16$), PXSAT manages to achieve a median success rate of 50.2% on PX applications. This is impressive because PX is only triggered on plateaus that are difficult to escape for local search.

Figure 4.6 visualizes the VIG and the decomposed recombination graph of the instance where Sparrow-PX has the largest performance gain. The original graph appears axisymmetric with two densely connected cores on each side and the connections between the two cores are sparser. PX successfully breaks one of the cores, leading to 842 components. One application of PX yields an instant improvement of 316 additional satisfied clauses, while Sparrow fails to discover any improving move in the last 72001 bit flips.

[20]A Spearman correlation of 1 results when the two variables are monotonically related.

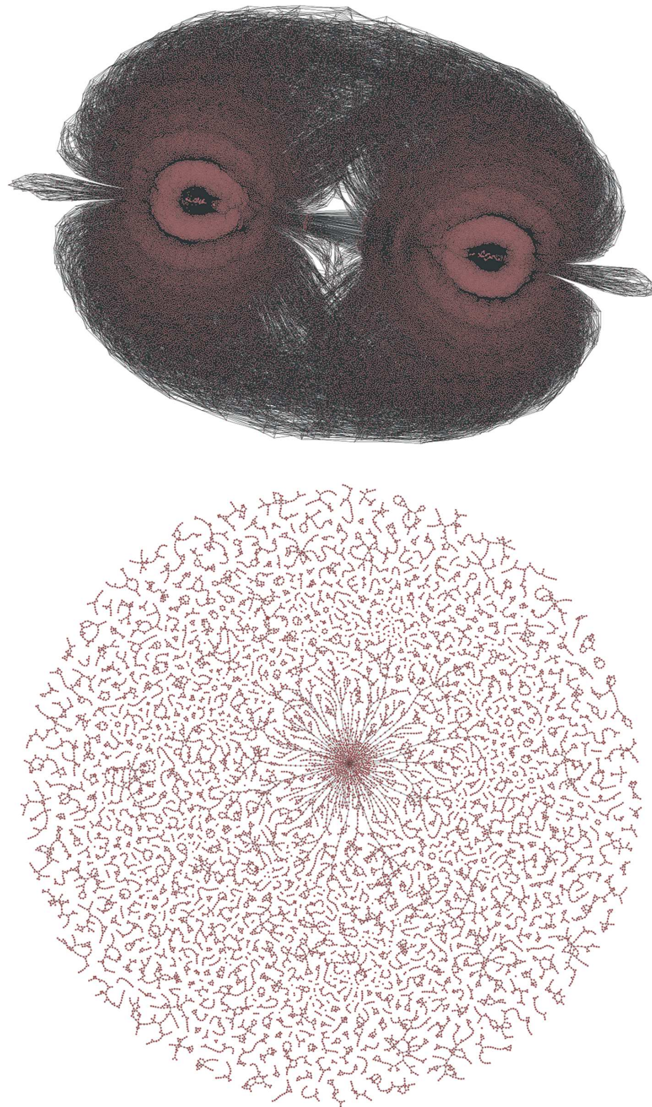


**Figure 4.6:** The VIG (top) and the decomposed recombination graph (bottom) for SAT_instance_N=111. In this instance, tunneling will return the best of $2^{842}$ solutions.

Using a force-directed graph layout algorithm[21], Figure 4.6 visualizes the VIG and the recombination graph of the instance where Sparrow-PX has the largest performance gain. The original graph appears axisymmetric with two densely connected cores on each side and the connections between the two cores are sparser. PX successfully breaks one of the cores, leading to 842 components. This single application of PX yields an instant improvement of 316 in evaluation, whereas Sparrow fails to discover any improving move in the last 72001 iterations.

[21] We use the library graph-tool (https://graph-tool.skewed.de/).

**Table 4.4:** Comparing AdaptG$^2$WSAT , Sparrow and their PXSAT versions with CCLS. "# Better (Worse) $\Delta sol$": number of instances where the PXSAT version has better (worse) average solution quality. "# Faster (Slower) $\Delta time$": number of instances where the PXSAT version has faster (slower) average solving time.

| Compared with CCLS | AdaptG$^2$WSAT | AdaptG$^2$WSAT-PX | Sparrow | Sparrow-PX |
|---|---|---|---|---|
| # Better $\Delta sol$ (Total) | 72 | 80 | 53 | 58 |
| # Better $\Delta sol$ (Sig) | 68 | 77 | 50 | 53 |
| # Worse $\Delta sol$ (Total) | 9 | 0 | 33 | 27 |
| # Worse $\Delta sol$ (Sig) | 7 | 0 | 27 | 22 |
| # Faster $\Delta time$ (Total) | 14 | 15 | 10 | 11 |
| # Slower $\Delta time$ (Total) | 4 | 4 | 3 | 3 |

Overall, $\check{lr}$ is very conservative predictor for the success of PXSAT. When $\check{lr} > 0$, PXSAT always improves the performance in the empirical study. When $\check{lr} < 0$, there is still a good chance of making a positive impact. PXSAT improves 35 out of 41 instances with $\check{lr} < 0$.

### 4.3.4 Competing with State-of-Art MAXSAT Solver

CCLS [167] is one of the best solvers designed specifically for MAXSAT, and has won several categories of the incomplete algorithms track of MaxSAT Evaluation 2013 to 2016. CCLS[22] is evaluated on the same benchmark for comparison, in order to access the *absolute performance* of PXSAT-equipped local search solvers.

Table 4.4 summarizes the results. CCLS has significantly better average solution quality than the original AdaptG$^2$WSAT and Sparrow on 7 instances and 27 instances, respectively. Thanks to PXSAT, AdaptG$^2$WSAT-PX consistently outperforms CCLS on every single instance tested, while Sparrow-PX takes the lead on five additional instances.

## 4.4 Conclusions

PXSAT employs a powerful recombination operators, Partition Crossover (PX), to exploit decomposability on application instances and to escape plateaus. PX uses common assignments

[22] As the source code is not publicly available, we used the binary obtained from http://lcs.ios.ac.cn/~caisw/Code/CCLS2015.

among local optima to decompose Variable Interaction Graphs (VIGs) into $q$ components. The best solution of $2^q$ candidate solutions can be greedily constructed in linear time. Empirical studies on an extensive set of application instances show PXSAT statistically significantly improves the performance of two best local search solvers, AdaptG$^2$WSAT and Sparrow, on application instances. The improvement in solution quality is as much as $80\%$.

We present theoretical analysis for highlighting the search efficiency of PXSAT as well as guiding the design of PXSAT. A theoretical performance model is developed to understand why and when PXSAT is useful. The model successfully predicts when PXSAT is likely to improve the performance. The model shows, given the same amount of time, the number of candidate solutions inspected by PXSAT is up to $10^{626}$ times more than the greedy operators in modern local search solvers.

# Chapter 5

# Exploiting Subproblem Constrainedness with Alternative Representations

The previous chapters exploit the decomposability of the variable interaction graph of application SAT instances for improving SLS solvers. This chapter investigates a structural property that is based on formula partitioning: subproblem constrainedness. We observe that, on some application SAT instance classes, the original problem can be partitioned into several subproblems, where each subproblems is highly constrained. Under Conjunctive Normal Form (CNF) representation, the high constrainedness of a subproblem is indicated by the fact that the set of clauses forming the subproblem are defined over only a small number of Boolean variables [170]. While subproblem constrainedness has been exploited in SS solvers before [29, 36], we propose to exploit it in SLS solvers using two alternative representations that can be obtained efficiently based on the canonical CNF representation.

The first alternative representation is *Conjunctive Minterm Canonical Form (CMCF)* [38]. The a transformation of CNF into CMCF can be accomplished using a two-step process: CNF clauses are first partitioned into disjoint blocks such that each block contains CNF clauses with shared Boolean variables. CNF clauses in each block are then replaced by *Minterm Canonical Form* (i.e., the set of all partial solutions), which is found by enumeration. The intuition behind CMCF representation is as follows. CNF representation is more compact for instances with low subproblem constrainedness, because fewer number of CNF clauses are required for representing the small number of constraints. On the contrary, for instances with high subproblem constrainedness, looking at the *complement* representation makes more sense. We show empirically that a simple Stochastic Local Search (SLS) solver based on CMCF can consistently achieve a higher success rate using fewer evaluations than the SLS solver WalkSAT on two representative classes of structured SAT problems.

The second alternative representation is *Minterm Interaction Graph (MIG)* that aims to refine CMCF by increasing the cardinality of the objective function. The MIG is a graph, where each represents a block and there is an edge between two vertices if they share at least one Boolean variable. Considering the pairwise interaction between blocks effectively increases the cardinality of the objective function over CMCF. With MIG representations, a straightforward SLS solver inspired by WalkSAT called *Gforce* solves semiprime factoring instances thousands of times faster than the highly optimized WalkSAT, scales better than sophisticated SLS solvers SAPS [39] and AdaptG$^2$WSAT [16], and compete well against the best SLS solver Sparrow [40] in recent competitions, in terms of the raw CPU time.

A major strength of CMCF and MIG is that they implicitly exploit the variable dependencies. Encoding structured problems into SAT problems often introduces *dependent variables* [72], whose values are defined by a Boolean function of other variables. These dependent variables are usually required by the well-known Tseitin encoding [73] to achieve linear size conversion of propositional logic formulas to CNFs. SS solvers can handle variable dependencies by propagating independent variables to dependent variables via unit propagation. On the contrary, SLS solvers takes longer to propagate dependencies by its iterative nature. Developing SLS techniques that can effectively handle variable dependencies has been considered to be a fundamental challenge in propositional reasoning and search [76]. With the minterms in CMCF and MIG, literals in a minterm are treated as a single entity, therefore CMCF and MIG link dependent Boolean variables with independent ones. The variable dependencies defined by arbitrary Boolean function are automatically and implicitly respected in satisfying minterms.

## 5.1 Related Work

Subproblem constrainedness has been studied by Amir and McIlraith [29, 36]. They decompose the set of clauses of a SAT instance into subproblems that are loosely related, order the subproblems in descending c/v ratio (clause-variable ratio), solve the subproblem using a SAT procedure according to the ordering, and finally join [171] the partial solutions to all subproblems following

the subproblem order to obtain all models to the original instance. However, joining all partial solutions to different subproblems, despite finding all models, can lead to state explosion [172]. In this chapter, we propose to use local search to find a compatible full assignment.

Previous research on handling variable dependencies in SLS solvers has focused on explicitly extracting a *predefined* set of Boolean circuit gates and searching only on independent variables. McAllester and Selman propose DAGSAT [72], which represents "AND" gates and "OR" gates as Directed Acyclic Graphs (DAGs) to encode the variable dependencies in a hierarchy. Pham, Thornton and Satter extend DAGSAT by also extracting "XNOR" gates and "XOR" gates [79] [80]. Our research differs from the dependency extraction research in two ways: 1) we do not require a predefined and limited set of Boolean circuit gates as the target for extraction. We instead encode the highly constrained subproblems using satisfying minterms, which can be *arbitrary* Boolean functions; 2) rather than explicitly searching on independent variables, we link independent variables and dependent variables using satisfying minterms and so variable dependencies are respected implicitly.

Roussel [173] proposed a transformation that uses a local model enumeration similar to ours. Roussel's encodes sets of clauses using prime implicants, while ours encodes them as satisfying minterms. However, Roussel did not show the utility of the new representation in search.

## 5.2 Exploit Subproblem Constrainedness using Conjunctive Minterm Canonical Form

SAT problem instances are usually defined in CNF: a conjunction of clauses $\mathcal{F} = \bigwedge_{c_i \in \mathbb{C}} c_i$, where $\mathbb{C}$ is the set of all clauses, each clause $c_i$ is a disjunction of literals $c_i = \bigvee_{l_j \in \mathbb{L}_i} l_j$, and each literal is either a Boolean variable $b_j$ or its negation $\overline{b_j}$. [23] The DIMACS format is the standard file format used to succinctly represent CNF instances [10]. In the DIMACS format, each line represents a

[23]In this chapter, italicized letters such as $v$ present single elements, capitalized bold font such as $\mathbf{V}$ represents a vector, capitalized blackboard font such as $\mathbb{V}$ represents a set and $|\mathbb{V}|$ represents the cardinality of the set $\mathbb{V}$.

clause with a "0" as the end-of-line delimiter. A clause is defined by listing the index of each positive literal, and the negative index of each negated literal.

Application instances often contain blocks of clauses consisting of a small set of variables with different combinations of signs. The pattern demonstrate that some subproblems are *highly constrained* in application instances. To illustrate this pattern, Snippet 1 shows a portion of a DIMACS formatted SAT encoding of the problem of factoring semiprime 1003. [24]

**Snippet 1** The 5th to 12th clauses ($c_5$ to $c_{12}$) of factoring-1003.cnf.

```
-123   24   33  0
-123  -24  -33  0
 123   24  -33  0
 123  -24   33  0
```

```
-24  -33   124  0
 24   33  -124  0
 24  -124  0
 33  -124  0
```

One can observe from Snippet 1 that the set of 4 clauses $M_1 = c_5 \wedge c_6 \wedge c_7 \wedge c_8$ contains exactly three Boolean variables $\{b_{123}, b_{24}, b_{33}\}$, and each clause has a different combination of signs on the three Boolean variables. The same pattern also applies to the other set of clauses: $M_2 = c_9 \wedge c_{10} \wedge c_{11} \wedge c_{12}$. Denote the formula in Snippet 1 as $\mathcal{F}'$, then $\mathcal{F}' = M_1 \wedge M_2$. This pattern indicates high subproblem constrainedness, which often appears in the SAT-encoded factoring instances. In fact, in many of the SAT-encoded structured instances, clauses appear to be generated such that those CNF clauses that share variables appear consecutively [170].

### 5.2.1 Sources for the Highly Constrained Subproblems

The observed highly constrained subproblems could be a coincidence or an artifact of some ordering decisions in the generation code, in which case it would not represent exploitable problem structure. By analyzing how SAT encoding is performed, we discover three possible sources for the highly constrained subproblems in application instances: 1) Tseitin encoding of propositional logic

[24] The SAT-encoded CNF instance is generated by ToughSAT [23].

formulas; 2) CNF encoding of Constraint Satisfaction Problem (CSP); 3) intrinsic loosely coupled components in the original problem.

**Tseitin Encoding**. Perhaps the most well-known and widely used method for converting logic circuit to CNF is Tseitin encoding [73]. It generates a linear number of CNF clauses by introducing a linear number of auxiliary variables, which avoids producing exponentially large CNFs. Consider the propositional formula $b_1 \rightarrow (b_2 \wedge b_3)$, its Tseitin encoding is $(b_2 \vee \overline{b_4}) \wedge (b_3 \vee \overline{b_4}) \wedge (\overline{b_2} \vee \overline{b_3} \vee b_4) \wedge (\overline{b_1} \vee b_4)$, where $b_4$ is an auxiliary variable introduced during the encoding process. The CNF formula exhibits the high subproblem constrainedness, i.e., four clauses containing the same four variables.

**CNF Encoding of CSP**. One way of encoding a structured problem as SAT is to first model it as a Constraint Satisfaction Problem (CSP) and then to encode the CSP as a SAT problem [32]. A finite-domain CSP has multi-value variables $v_i$, each associated with a finite domain $dom(v_i)$. CSP constraints describe prohibited combinations of assignments. The most natural and widely used SAT encoding of CSP is the *direct encoding*, in which a SAT Boolean variable $b_{v,i}$ is true if and only if the CSP multi-valued variable $v$ is assigned to $i$. Suppose we have a CSP variable $v_1$ where $dom(v_1) = \{1, 2, 3\}$. The direct SAT encoding enforces the exactly-one constraint on the Boolean variables that encode the same CSP variable: at any given time, any CSP variable are assigned to one single value. In the example, exactly one of $b_{1,1}, b_{1,2}, b_{1,3}$ is set to true. Encoding the exactly-one constraint on $v_1$ results in the clauses $(b_{1,1} \vee b_{1,2} \vee b_{1,3}) \wedge (\overline{b_{1,1}} \vee \overline{b_{1,2}}) \wedge (\overline{b_{1,1}} \vee \overline{b_{1,3}}) \wedge (\overline{b_{1,2}} \vee \overline{b_{1,3}})$. This set of four clauses contains just 3 variables. Again, the subproblem is highly constrained, in the sense that the set of four clauses can only be satisfied in 3 ways.

**Intrinsic loosely coupled components in the original problems**. Besides the CNF encoding process that introduces highly constrained subproblems, the source problem may already include intrinsic loosely coupled components, in which each component (subproblem) are highly constrained. The original problem may contain multiple knowledge databases that have overlap in content [29]. Each knowledge database as a subproblem is highly constrained. One prominent example for such case is the commonsense theories found in the DARPA High Performance Knowledge Base (HPKB) program [36].

|  | $b_{123}$ | $b_{24}$ | $b_{33}$ |
|---|---|---|---|
| $s_{1,1}$ | 0 | 0 | 0 |
| $s_{1,2}$ | 0 | 1 | 1 |
| $s_{1,3}$ | 1 | 1 | 0 |
| $s_{1,4}$ | 1 | 0 | 1 |

**(a)** $M_1$

|  | $b_{24}$ | $b_{33}$ | $b_{124}$ |
|---|---|---|---|
| $s_{2,1}$ | 0 | 0 | 0 |
| $s_{2,2}$ | 1 | 0 | 0 |
| $s_{2,3}$ | 0 | 1 | 0 |
| $s_{2,4}$ | 1 | 1 | 1 |

**(b)** $M_2$

**Figure 5.1:** Satisfying minterms (the set of all partial solutions) to $M_1$ and $M_2$ of $\mathcal{F}'$ shown in Snippet 1, respectively. Each row represents a satisfying minterm to the block. A "1" under Boolean variable $b_i$ means the $b_i$ is set true in the partial solution, otherwise $b_i$ is set false; $s_{i,j}$ is the $j$th partial solution to $M_i$.

### 5.2.2 Exploiting High Subproblem Constrainedness: From CNF to CMCF

A number of application SAT instances exhibit high subproblem constrainedness in the way that CNF instances are generated. We next explore how this pattern can be exploited. To illustrate the idea, we enumerate satisfying minterms (partial solutions) to the blocks (groups of clauses) found in Snippet 1. In figure 5.1 clauses $c_5, c_6, c_7, c_8$ are grouped into block $M_1$; clauses $c_9, c_{10}, c_{11}, c_{12}$ clauses are grouped into block $M_2$. Each block induces a highly constrained subproblem. A closer look at the two blocks reveals that $M_1$ is an XOR gate $b_{123} = XOR(b_{24}, b_{33})$, and $M_2$ is an AND gate $b_{124} = AND(b_{24}, b_{33})$. The AND gate defines $b_{124}$ as a dependent variable whose value is determined by $b_{24}$ and $b_{33}$. For the XOR gate, any one of the three Boolean variables can be considered as the dependent variable in terms of the other two. Instead of flipping one Boolean variable to satisfy one clause, we can now change multiple variables at a time to guarantee *satisfying all* of the clauses in the block. In this way, although we are not looking for any specific kinds of variable dependencies, dependencies are *automatically and implicitly respected*.

In this work, two steps are taken for transforming CNF into CMCF: *formula partitioning* which partitions the set of all clauses into disjoint blocks; *minterm enumeration* where each block is solved optimally using enumeration.

## Formula Partition

**Definition 4** (Partition)**.** *A partition over a set is a grouping of the set's elements into non-empty disjoint subsets.*

Denote the set of all clauses as $\mathbb{C}$, formula partition applies a partition operation $P$ to group $\mathbb{C}$ into disjoint subsets. For instance, the partition for $\mathcal{F}'$ in Snippet 1 is $\{\{5, 6, 7, 8\}, \{9, 10, 11, 12\}\}$. A *block* $M_i$ is a conjunction of the clauses in a proper subset $\mathbb{C}_i \subsetneq \mathbb{C}$:

$$M_i = \bigwedge_{c_j \in \mathbb{C}_i} c_j, \tag{5.1}$$

where $\bigvee \mathbb{C}_i = \mathbb{C}$ and $\mathbb{C}_i \bigwedge_{i \neq j} \mathbb{C}_j = \emptyset$. Each block $M_i$ can be considered as a subproblem. The original problem is therefore partitioned into $|P|$ subproblems with overlaps between them. The quality of a partition is assessed based on the degree of overlap between the subproblems, which is further measured by the number of variables shared by different blocks. The goal is to partition $\mathbb{C}$ such that the overlap between subproblems is minimal.

In the current work, we use a straightforward greedy partitioning method as a starting point. Suppose $X$ is a clause or a set of clauses, $Vars(X)$ represents the set of Boolean variables in $X$. We limit the maximum of number of Boolean variables in each block to be 6, namely, $max(|Vars(M_i)|) = 6$. On instances that have clauses longer than 6, we introduce auxiliary variables to break the clauses until they are within the limit. Initially, a new empty block $M_1$ is created. It iterates through all clauses, if the current block $M_i$ can take the clause $c_j$ without breaking the limit, $c_j$ is added to the current block. Otherwise, a new $M_{i+1}$ is created to take $c_j$.

## Minterm Enumeration

**Definition 5** (Minterm [174])**.** *Given a Boolean function $\mathcal{F}$, a minterm for $\mathcal{F}$ is a the conjunction of literals, in which every variable in $\mathcal{F}$ appears exactly once in either its original or negated form.*

Any Boolean function $\mathcal{F}$ can be designated using the disjunction of the minterms that evaluates $F$ to true. In other words, the satisfying minterms are the solutions to $\mathcal{F}$. Enumerating partial solutions to a block $M_i$ *encodes* the subproblem in terms of satisfying minterms. From this perspective, a block $M_i$ can also be defined as disjunction of satisfying minterms:

$$M_i = \bigvee_{s_j \in \mathbb{S}_i} s_j, \tag{5.2}$$

where $\mathbb{S}_i$ is set of all satisfying minterms to $M_i$. Having the two equivalent definitions of $M_i$ as shown in equation 5.1 and equation 5.2, we can then *replace* blocks that are defined using the first definition of $M_i$ (as in CNF) with its equivalent second definition. Now for every block, replace the conjunction of clauses $\bigwedge_{c_j \in \mathbb{C}_i} c_j$ with disjunction of satisfying minterms $\bigvee_{s_j \in \mathbb{S}_i} s_j$. We call the resulting representation a *Conjunctive Minterm Canonical Form (CMCF)*, in the sense that it is a conjunction of Boolean functions defined by disjunction of satisfying minterms.

**Definition 6** (CMCF)**.** *Given a partition $P$ over the set of all clauses set $\mathbb{C}$ of formula $\mathcal{F}$, the Conjunctive Minterm Canonical Form (CMCF) of $\mathcal{F}$ is $\bigwedge_{M_i \in P} (\bigvee_{s_j \in \mathbb{S}_i} s_j)$, where $M_i$ is a nonempty subset of the partition and $\mathbb{S}_i$ is the set of satisfying minterms to $M_i$.*

### 5.2.3 Constraint Propagation over Minterms

The use of minterms allows straightforward constraint propagation. We introduce two forms of constraint propagations: *Value Constraint Propagation* and *Equivalency Constraint Propagation*.

**Value Constraint Propagation.** A Boolean variable $b_i$ that has a consistent truth value in a block $M_j$ indicates that it is the only way $b_i$ can be set to satisfy $M_j$, which is also a necessary condition for satisfying the entire formula $\mathcal{F}$. We can then propagate the assignment to other blocks.

Take Figure 5.2 for an example, $b_3$ is assigned to 0 consistently across all satisfying minterms in $M_1$, which suggests that $b_3$ must be assigned to 0 in a solution (if it exists) to $F$. We can therefore fix the $b_3$ to 0, and propagate the assignment to other blocks that also contain $b_3$. Later we discover

that $M_2$ also contains $b_3$ and it has a minterm that contradicts the assignment $b_3 = 0$. The minterm $b_3 \wedge b_4 \wedge \overline{b_5}$ can thus be eliminated.

$M_1$

| $b_1$ | $b_2$ | $b_3$ |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 0 |

$\Longrightarrow M_2$

| $b_3$ | $b_4$ | $b_5$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| ~~1~~ | ~~1~~ | ~~0~~ |

**Figure 5.2:** An example illustrating value constraint propagation. Each row represents a satisfying minterm to the block.

**Equivalency constraint propagation.** When two Boolean variables $b_i$ and $b_j$ satisfy the equivalency $b_i \iff b_j$ or negative equivalency $b_i \iff \overline{b}_j$ constraint in a block $M_k$, $b_j$ can be eliminated by replacing all occurrences of $b_j$ with $b_i$ or $\overline{b}_i$, which can further lead to possible elimination of incompatible minterms.

Take Figure 5.3 for an example, $M_1$ exhibits one equivalency constraint $b_3 \iff b_1$ and one negative equivalency constraint $b_2 \iff \neg b_1$, which can be used to eliminate $b_2$ and $b_3$. In this case, $b_2$ in $M_2$ is replaced with $\neg b_1$, flipping the truth values assigned to $b_2$ in the minterms in $M_2$. $M_2$ then contains two columns of $b_1$. Any incompatible assignment to $b_1$ can be eliminated, i.e., $b_1$ can not be true and false simultaneously. The incompatible minterm $b_1 \wedge \overline{b_1} \wedge b_4$ is therefore eliminated.

$M_1$

| $b_1$ | $b_2$ | $b_3$ |
|---|---|---|
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |

$\xrightarrow[b_3 \iff b_1]{b_2 \iff \neg b_1}$ $M_2$

| $b_1$ | $b_2$ | $b_4$ |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |

$M_2$

| $b_1$ | $b_1$ | $b_4$ |
|---|---|---|
| ~~0~~ | ~~1~~ | ~~1~~ |
| 1 | 1 | 1 |
| 0 | 0 | 0 |
| 1 | 1 | 0 |

**Figure 5.3:** An example illustrating equivalency constraint propagation. Each row represents a satisfying minterm to the block.

Eliminating minterms using one constraint propagation potentially opens up new opportunities for the application of the other constraint propagation. In the current work, the two constraint propagations are repeated alternatively, until there is no change in CMCF representation of $\mathcal{F}$.

### 5.2.4 Local Search over CMCF

Rather than explicitly natural joining minterms from every block as in [36], which can lead to state explosion, we employ a simple local search, *CMCF-LS*, to find a valid combination of minterms. CMCF-LS is an SLS analogous to WalkSAT [15] to search over CMCF representation. We chose WalkSAT, because it has a straightforward form, and is very effective at solving uniform random SAT instances. It is a good basis for designing SLS on the new representation that exploits subproblem constrainedness. The search space explored by CMCF-LS is the different from the one explored by WalkSAT. This leads to the following designs in CMCF-LS.

**Representation of a Candidate Solution**. With the CMCF representation, every block $M_i$ contributes exactly one minterm $s_{i,j}$, where $s_{i,j} \in M_i$. A candidate solution in the search space of CMCF-LS is a vector of minterms $s_{i,j}$, where $i \in [1, |P|]$. SAT solving under CMCF representation is to find a combination of minterms, one from from each block, that have a consensus on setting all Boolean variables.

For a given block $M_i$, there are $|\mathbb{S}_i|$ choices of minterms. Any combination of choices drawn from all blocks is a candidate solution. Multiplying the number of choices over all blocks gives the size of the search space under CMCF representation, $\prod_{M_i \in P} |\mathbb{S}_i|$. To reduce the search space, we apply the two constraint propagations in Subsection 5.2.3 as a preprocessing step to eliminate minterms in blocks, effectively reducing $|\mathbb{S}_i|$.

**Consensus Constraint**. A minterm $s_{i,j}$ from a block $M_i$ *votes* for the truth assignment for the Boolean variables in $M_i$. Different blocks that share a Boolean variable can possibly vote the variable differently. We call this *discrepancy* on Boolean variables. For example, the candidate $(s_{1,2}, s_{2,2})$ of $\mathcal{F}'$ from Figure 5.1 votes for $b_{123} = 0, b_{24} = 1, b_{33} = 1$ by $s_{1,2}$, and votes for $b_{24} = 1, b_{33} = 0, b_{124} = 0$ by $s_{2,2}$. The only discrepancy is at $b_{33}$, since $b_{33}$ receives one positive vote from $s_{1,2}$ and one negative vote from $s_{2,2}$. Each Boolean variable $b_i$ imposes a *Consensus Constraint* on candidate solutions, such that $b_i$ should only receives a single kind of vote, either true or false. The number of consensus constraints thus equals to the number of Boolean variables.

A candidate solution without violating any consensus constraint is a *valid* solution, i.e., reaching consensuses on setting all Boolean variables.

**Evaluation Function**. The evaluation of a candidate solution goes through every Boolean variable to collect the true votes and false votes from all blocks. For a Boolean variable $b_i$, we use the smaller of two counts to represent the degree of discrepancy, since it indicates the minimal number of minterms that need to change to satisfy the consensus constraint on $b_i$. The evaluation is then the summation of the smaller vote counts over all Boolean variables. We call this evaluation function *MinVote*. The process for evaluating candidate solution $(s_{1,2}, s_{2,2})$ of $\mathcal{F}'$ from Figure 5.1 is shown in Table 5.1. The evaluation of $(s_{1,2}, s_{2,2})$ is therefore $0 + 0 + 1 + 0 = 1$.

**Table 5.1:** Demonstration of Calculation of MinVote evaluation of candidate solution $(s_{1,2}, s_{2,2})$ for $\mathcal{F}'$ as shown in Figure 5.1.

| Boolean Variables | $b_{123}$ | $b_{24}$ | $b_{33}$ | $b_{124}$ |
|---|---|---|---|---|
| True Votes | 0 | 2 | 1 | 0 |
| False Votes | 1 | 0 | 1 | 1 |
| MinVote | 0 | 0 | 1 | 0 |

**SLS Operators**. In each iteration of WalkSAT [15], it first selects an unsatisfied clause, then either makes a random move that flips a random variable in the clause with probability $p$, or with probability $(1 - p)$ makes a greedy move that flips the variable that results in the highest improvement in the evaluation function.

CMCF-LS works in a way similar to WalkSAT, it first selects a Boolean variable $b_i$ that has a discrepancy. With probability $p$, CMCF-LS makes a *random move* that forces all blocks containing $b_i$ to reach a consensus state on a random truth assignment; with probability $(1 - p)$, CMCF-LS makes a *greedy move* that select the minterm from all blocks containing $b_i$ such that it yields the highest improvement in the evaluation function.

### 5.2.5 Empirical Results on CMCF-LS

The transformation from CNF to CMCF alters the search space for SLS. The CMCF-based constraint propagations then reduce the overall size of the CMCF search space. The paramount issue is whether this process leads to an improvement in the success of SLS. At this point, we have a straightforward implementation of our solver as a proof of concept, and so focus our evaluation on demonstrating the utility of the approach. We consider two questions:

- Does the preprocessing step using CMCF-based constraint propagations effectively reduce the size of the search space?
- Can CMCF-LS show an improvement over the CNF-based solver WalkSAT?

## Problem Instances

Two classes of structured instances are used in our empirical study: Semiprime Factoring and Parity Learning. The CNF encodings of both exhibit a high subproblem constrainedness. Factoring large semiprimes is a difficult problem with no known method for quick solution [175]. In fact, its difficulty is the basis of the assumed security for RSA public key cryptography. We randomly generated 50 pairs of distinct primes and multiplied each pair to obtain 50 semiprimes. The ToughSAT generator (with "Factoring 2" as problem type) [23] is used to encode the semiprime factoring problem.

The Minimal Disagreement Parity learning problem is a class of hard satisfiability problems [83]. Given a $m \times n$ matrix of Boolean variables $b_{ij}$ ($i \in [1, m], j \in [1, n]$), a vector $\mathbf{y} = (y_1, \ldots, y_m)$ and an error tolerance level $k$, the parity learning problem is to find a vector of Boolean variables $\mathbf{a} = (a_1, \ldots, a_n)$ such that $|i : parity(\mathbf{a} \cdot \mathbf{x}_i) \neq y_i| \leq k$. These problems are hard for SLS, due to many local minima which are close to the optimum [76]. The encoding of the parity constraints creates a long chain of variable dependencies [71]. We test on the five uncompressed 8-bit parity learning instances available from SATLIB [176].

**Table 5.2:** Problem instance sizes as measured by number of variables, number of clauses for CNF, number of blocks for CMCF encodings, and average number of CNF clauses per Block.

| | Variables | CNF Clauses ($|\mathbb{C}|$) | CMCF Blocks ($|P|$) | Clauses per Block |
|---|---|---|---|---|
| factoring | 347.2(89.9) | 1740.7(474.7) | 169.9(44.4) | 10.24 |
| par8-1 | 350 | 1149 | 144 | 7.97 |
| par8-2 | 350 | 1157 | 144 | 8.03 |
| par8-3 | 350 | 1171 | 145 | 8.07 |
| par8-4 | 350 | 1155 | 142 | 8.13 |
| par8-5 | 350 | 1171 | 146 | 8.02 |

# Search Space Reduction

The search space size for CNF encoding of SAT is familiar. But what happens with our proposed transformation? To assess this, we collected data on instance size after the transformation and the constraint propagations. To show the overhead, we also collected data on the CPU time required for the transformation.

**How the problem size changes**. Table 5.2 shows the problem sizes as measured in number of variables and number of clauses for the original CNFs and the CMCF transformations. The semiprime factoring instances are captured by the mean and standard deviation (with form "mean (std)") across the 50 instances; each parity instance is listed separately. The number of variables remains the same in each representation. The straightforward partition method works reasonably well in practice, resulting in good compression rate (from 7.97 to 10.24) over CNF clauses. It suggests that the partition method is able to group many clauses that involve at most 6 Boolean variables together in one block, which reveals the high subproblem constrainedness in the CNF encoding.

The partition step organizes the CNF clauses into blocks. Then, CMCF-based constraint propagations reduce the number of candidate solutions by eliminating minterms. For the semiprime factoring instances, Figure 5.4 shows the size of the search space after partitioning (before.CMCF.CP) and then after constraint propagation (after.CMCF.CP). The parity instances show similar reductions in each instance, e.g., from $1.43e + 130$ to $5.07e + 45$ for par8-1, which is substantially smaller than

**Figure 5.4:** Search space size on log-10 scale before and after CMCF-based constraint propagations on semiprime factoring problem instances.

the CNF search space $2^{350} = 2.29e + 105$. Constraint propagation leads to substantial search space reduction.

**How much CPU time is required**. Figure 5.5 shows the runtime overhead of CMCF-based constraint propagations, as measured in CPU seconds, as a function of the number of variables in the original CNF representation for the semiprime factoring problems. The CPU times are all very small, with the highest runtime overhead being 0.00129 seconds on the factoring-2419 instance with 470 Boolean variables. The code was run on shared machines which produced some of the variability in the times. An adjusted R-square of 0.8241 suggests a strong linear trend in problem size. The five parity learning instances required less than 0.0005 CPU seconds. On these problems, CMCF-based constraint propagations appears to have low runtime overhead.

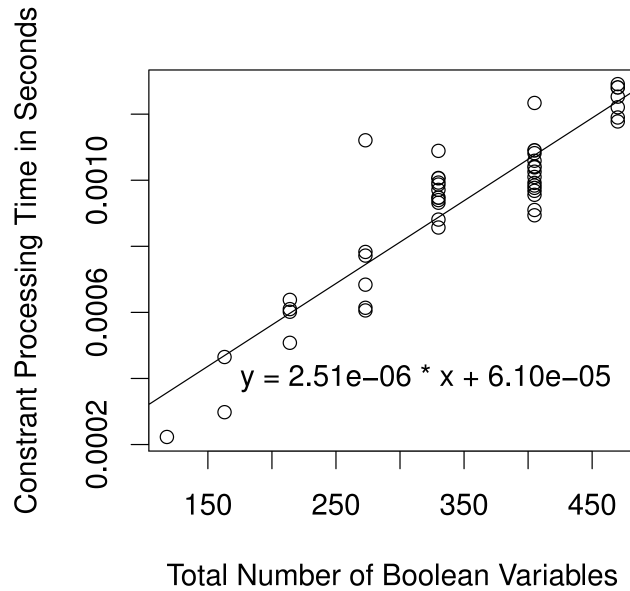



**Figure 5.5:** Runtime overhead for CMCF-based constraint propagations on semiprime factoring problem instances as a function of the number of variables in the original CNF version. The line is a linear least squared fit with adjusted R-squared of 0.8241.

# SLS Comparison

There are many SLS solvers for SAT. We chose WalkSAT for the comparison because it has a straightforward algorithm, has been influential in the design of subsequent SLS solvers and provided the basis for CMCF-LS. We implemented CMCF-LS from scratch in C++ and used the UBCSAT [17] implementation of WalkSAT, which is considered to be one of the most optimized implementations. We ran both CMCF-LS and WalkSAT for 50 independent runs with different seeds; each run allows for 100 million evaluations.

CMCF-LS and WalkSAT are compared on two aspects: effectiveness and cost of search. SLS solvers prove satisfiable for a given satisfiability problem instance by finding a satisfying assignment. The *success rate* (% of runs in which the SLS solver found a solution) is reported as the metric for effectiveness. CMCF-LS explores a neighborhood of size different from WalkSAT's. In each greedy pick of CMCF-LS, it examines all minterms in all the blocks that contain unsigned Boolean variables, and selects a minterm from a block with the highest evaluation. WalkSAT only evaluates all Boolean variables in an UNSAT clause to find the one with the highest evaluation. CMCF-LS is obviously more expensive than WalkSAT in performing the greedy pick. The random picks for

both require no evaluation of any candidate solution. For a fair comparison, we use *the number of evaluated candidate solutions* and *the CPU time* as the metric for the cost of search.

**Effectiveness of Search**. It is difficult to predict how an alternative or more compact representation will impact solver performance. For example, using a preprocessor such as SatElite [102] to reduce the problem size does not always translate into a speedup in the solver.

Figure 5.13 reports the success rate of 50 independent runs of both CMCF-LS and WalkSAT over the semiprime factoring instances, which are ordered by the semiprime values. The two curves generated by local polynomial regression fitting [177] show a clear advantage of CMCF-LS over WalkSAT; CMCF-LS consistently solves instances of semiprime smaller than 1000 in almost all of the runs, while WalkSAT's success rate degrades more quickly with semiprimes higher than 500. Once the semiprimes surpass 1500, the success rate of WalkSAT drops below 0.15 with WalkSAT almost never solving instances for semiprimes larger than 2000. CMCF-LS, on the other hand, solves 9 instances of factoring semiprimes over 1000 and still solves the three largest problems (of semiprimes 3127, 3233 and 3599) in at least one of the runs.

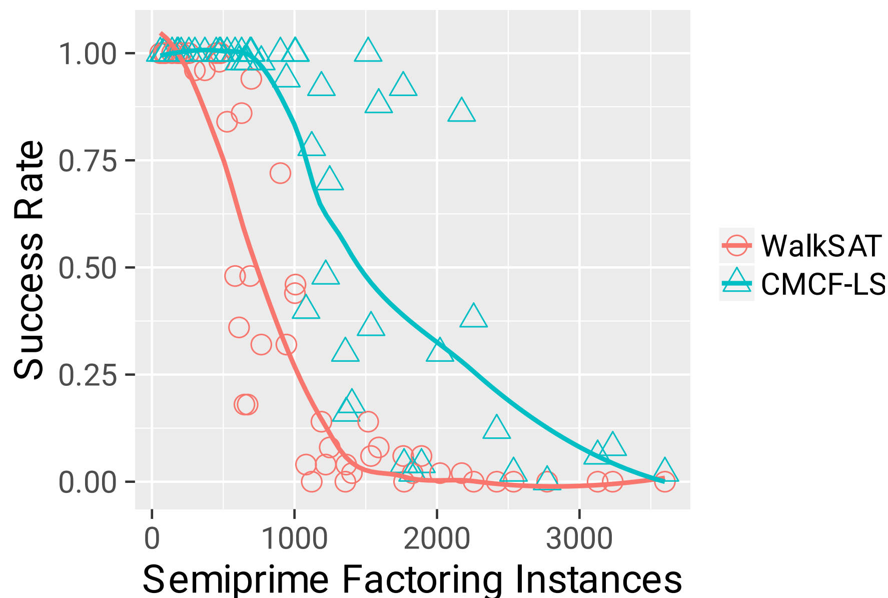



**Figure 5.6:** Success Rate on 50 Semiprime Factoring Problem Instances. The two trend curves are constructed using local polynomial regression fitting.

Figure 5.7 shows the success rate for the five parity instances. CMCF-LS consistently solves all 5 instances in all runs. In contrast, the highest success rate of WalkSAT is 0.68 on par8-1. WalkSAT can solve par8-5 in only one run.

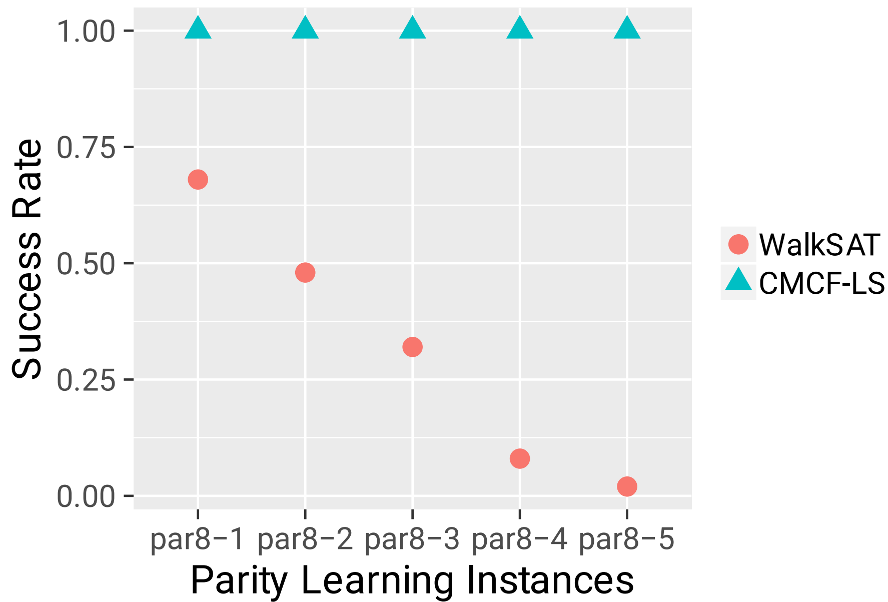



**Figure 5.7:** Success rate for the five parity learning instances.

**Number of Evaluated Candidate Solutions**. Although the SLS solvers are given up to 100 million evaluations, each run finishes when a satisfiable variable assignment is found. Figure 5.8 presents the number of evaluations required by each of the solvers (CMCF-LS in blue and WalkSAT in red) on each of the semiprime factoring instances. On the three smallest instances (namely 58, 82 and 94), WalkSAT actually takes fewer evaluations, as measured by the median, than CMCF-LS. The advantage flips on the first instance larger than 100 (namely 141). CMCF-LS does equal to or better than WalkSAT on all instances larger than 100. The shape of the two curves after 100 is also similar, which makes sense since the algorithmic framework of random and greedy moves is common to both.

For the parity learning problems, there is only one instance (par8-1) for which WalkSAT has a median lower than the 100 million cutoff. On par8-1, the median number of evaluations spent by WalkSAT is more than 95 times that of CMCF-LS ($\frac{52,970,392}{5,534,32} = 95.7$). On par8-5, the best (and the

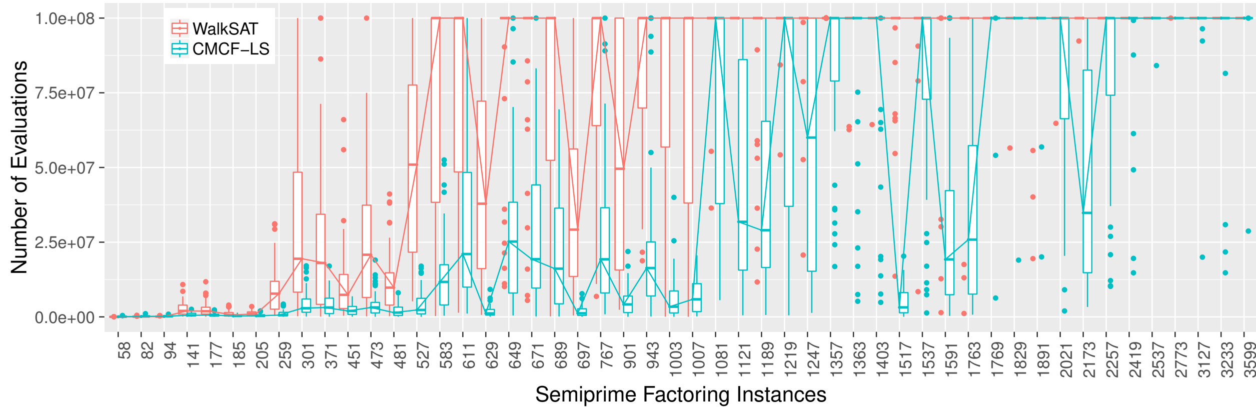



**Figure 5.8:** Number of Evaluations spent by CMCF-LS and WalkSAT on Semiprime Instances. Boxplot shows the distribution over 50 runs. Lines connect the medians.

only successful) run of WalkSAT needs 3,983,883 evaluations, which is over $150\times$ more than that of CMCF-LS (26,521 evaluations).

**CPU Time**. Even though substantial advantage of CMCF-LS over the base solver WalkSAT in term of number of evaluations has been observed, we doubt this advantage will also turn into improvement in terms of the raw CPU time. WalkSAT is highly optimized by design [15], and continues to receive performance optimization updates due to the fact that it is the basis for many best performing SLS solvers.

Figure 5.10 reports the CPU time comparison between CMCF-LS and WalkSAT on semiprime factoring instances. We observe that WalkSAT consistently takes less CPU time than CMCF-LS, confirming our speculation. CMCF-LS, in its current unoptimized form, cannot match the raw performance of WalkSAT on semiprime factoring instances.

Figure 5.11 represents the CPU time comparison between CMCF-LS and WalkSAT on parity learning instances. Surprisingly, the unoptimized CMCF-LS can match the performance of the highly optimized WalkSAT, even in terms of CPU time. On par8-1 for instance, the median CPU time of CMCF-LS is 1.476 seconds, which is only 40% of 3.635 seconds by WalkSAT.

The empirical results showed the potential of CMCF-LS, even in its present straightforward and unoptimized form. The simple partitioning method can group many related clauses together in one group. The constraint preprocessing under the CMCF representation eliminated substantial

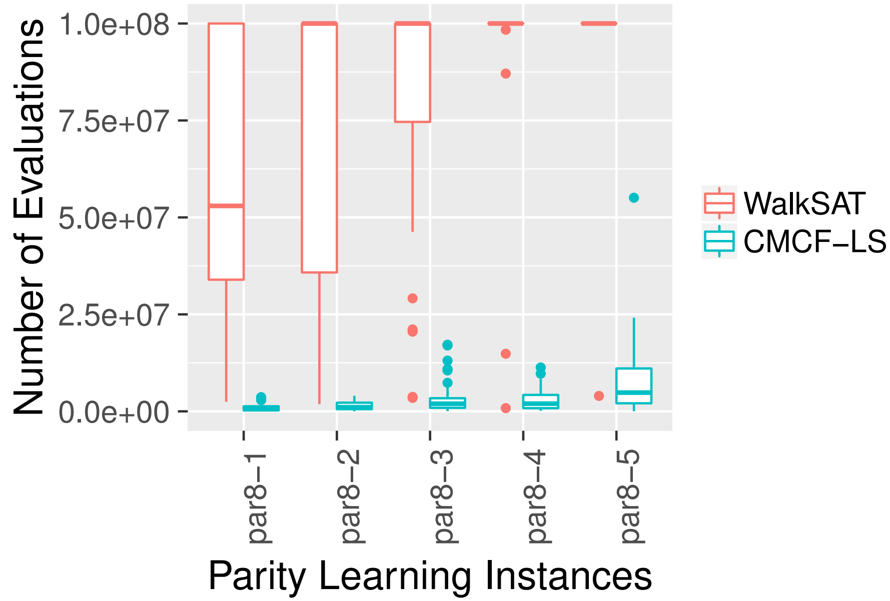



**Figure 5.9:** Number of Evaluations for CMCF-LS and WalkSAT on parity instances. Boxplot shows the distribution over 50 runs.

amount of minterms, enabling drastic reduce in search space in the preprocessing step. Moreover, the reduced search space in CMCF encoding translated into fewer number of evaluation and higher success rate of SAT solving on two classes of application SAT instances. Finally, the unoptimized CMCF-LS was able to match the raw performance of the highly optimized WalkSAT, thanks to CMCF-LS' superior representation.

## 5.3 Minterm Interaction Graph for Improving CMCF

Although CMCF showed significant promise in the previous section, it is far from being perfect. We ultimately hope to make the alternative representation-based local search solver beat not only the base solver WalkSAT [15] in terms of raw CPU time, but also the best performing SLS solver on application instances, like AdaptG$^2$WSAT [16]. In this section, we focus on two targets to improve upon CMCF. The first target is to refine the CMCF representation by increasing the cardinality of the evaluation function, so that the plateaus would be less troublesome for local search. The second target is to reduce the CPU time of the alternative representations by sourcing optimization techniques from CNF representation.

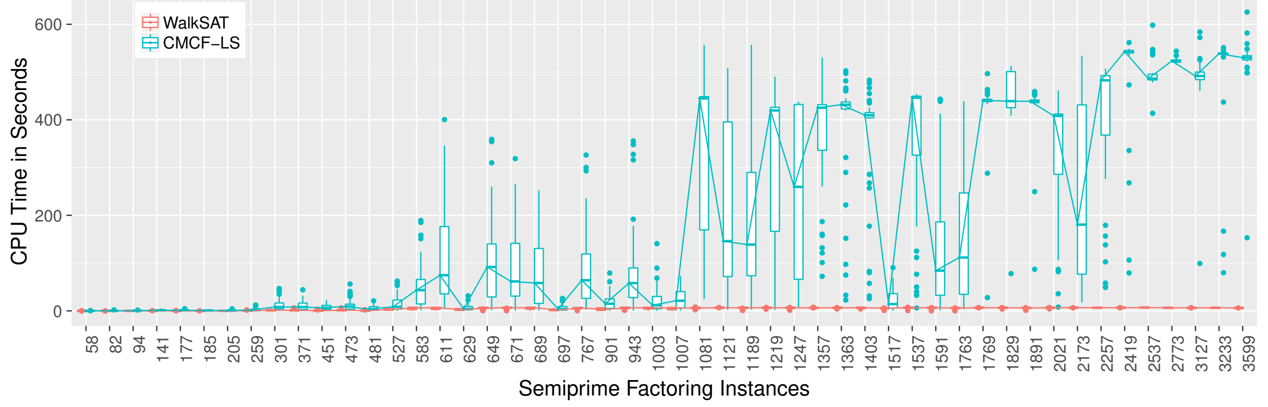



**Figure 5.10:** CPU Time spent by CMCF-LS and WalkSAT on Semiprime Instances. Boxplot shows the distribution over 50 runs. Lines connect the medians.

### 5.3.1 Increasing the Cardinality of Evaluation Function

One prominent issue that makes local search on SAT hard is the neural landscape that leads to extensive search on plateaus, which dominates the running time of SLS solvers [134]. One way to alleviate the neutrality of search landscape is to increase the cardinality of the evaluation. Increasing the cardinality of the evaluation in expectation gives more gradient information on plateaus, and potentially leads to faster escape from plateaus.

## Theory

Given a function $f$ with discrete outputs, the *cardinality* of $f$ is the number of its distinct outputs. We first study the cardinality of the MinVote evaluation function in CMCF-LS:

**Lemma 1.** *Let $\kappa_i$ be the number of blocks that contains a given Boolean variable $b_i$, the cardinality of the MinVote evaluation function in CMCF-LS is* $\frac{1}{2}\sum_{b_i \in \mathbb{B}} \kappa_i$.

*Proof.* $b_i$ receives $\kappa_i$ votes during evaluation, divided into $p_i$ positive votes and $n_i$ negative votes, where $p_i + n_i = \kappa_i$. Consider the MinVote evaluation function $min(p_i, n_i)$ for $b_i$. If $p_i \leq n_i$, $min(p_i, n_i) = p_i$, maximum of $p_i$ is reached when $p_i = n_i = \frac{\kappa_i}{2}$. This applies similarly to the case where $p_i \geq n_i$. Therefore, maximum of the MinVote evaluation $min(p_i, n_i)$ for is $\frac{\kappa_i}{2}$. Summing over the maximum for each variable, the maximum output for MinVote evaluation function is $\frac{1}{2}\sum_{b_i \in \mathbb{B}} \kappa_i$.

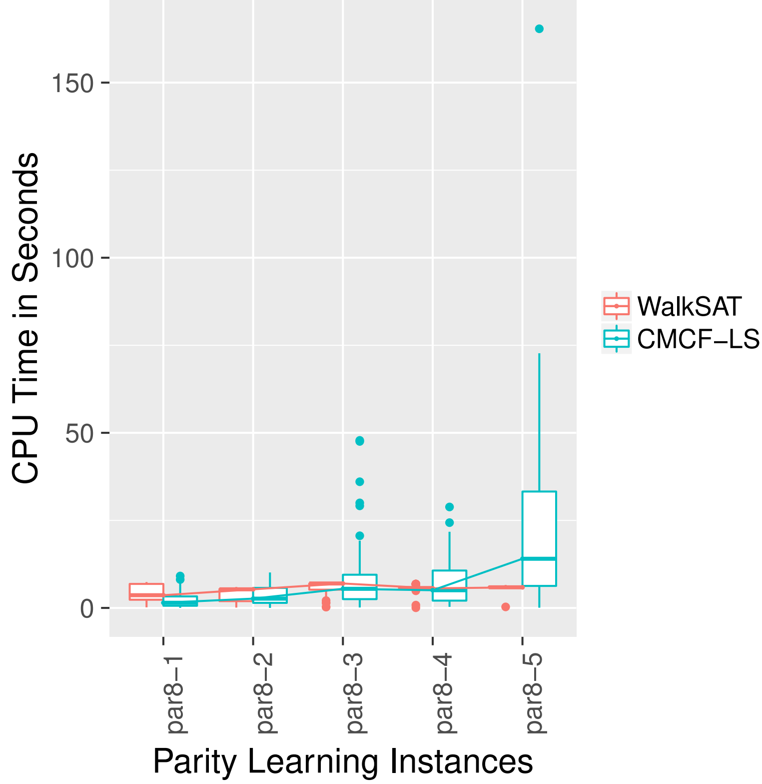



**Figure 5.11:** CPU Time spent by CMCF-LS and WalkSAT on parity instances. Boxplot shows the distribution over 50 runs.

Since the MinVote evaluation function can output any integer between 0 and the maximum value $\frac{1}{2}\sum_{b_i \in \mathbb{B}} \kappa_i$, its cardinality is $\frac{1}{2}\sum_{b_i \in \mathbb{B}} \kappa_i$ (omitting the constant term 1). $\square$

The result in Lemma 1 is in line with the intuition that the maximum of evaluation function is reached when there are equal number of positive votes and negatives.

We now consider the consensus constraints between interacting minterms that share Boolean variables, which gives rise to provably increased cardinality of evaluation function. We call two blocks, $M_i$ and $M_j$, *interacts* if the two blocks that they share some Boolean variable. We propose a new, refined representation: Minterm Interaction Graph[25].

[25]It is called "Minterm Interaction Graph" rather than "Block Interaction Graph". This is because "minterm" is a well-known and unambiguous term, while "block" is a more general term that can be used in different circumstances. Moreover, any two minterms, $s_i$ from $M_i$ and $s_j$ from $M_j$, interact $\iff$ $M_i$ and $M_j$ interact.

**Definition 7** (Minterm Interaction Graph (MIG)). *Minterm Interaction Graph is a graph, in which each vertex represent a block and an edge is established between two vertices if they share at least one Boolean variables.*

The representation of a candidate solution in the MIG remains the same as in CMCF. One minterm is selected from each block. The consensus constraint is now transferred onto the edges in the MIG. An edge that connects minterms $s_i$ and $s_j$ *satisfies* the consensus constraint, if the shared Boolean variables between $s_i$ and $s_j$ are assigned consistently. Notice that there can be multiple Boolean variable shared across two interacting blocks, even though the goal of partitioning method is to minimize the number of shared Boolean variables between blocks and to reduce the interactions between subproblems. Under the MIG representation, the evaluation function, called *EdgeCount*, is defined as the number of edges that violate the consensus constraint.

**Theorem 7.** *Let $\kappa_i$ be the number of blocks that contains a given Boolean variable $b_i$, and suppose the number of Boolean variables shared by any two blocks is bounded by a constant $\sigma$, the cardinality of the EdgeCount evaluation function for MIG is at least $\dfrac{1}{4\sigma}\displaystyle\sum_{b_i \in \mathbb{B}} \kappa_i^2$.*

*Proof.* The vertices (blocks) that containing any given Boolean variable $b_i$ forms a clique of size $\kappa_i$ in MIG; this is immediate from Definition 7. Let $V_i$ be the number of violated consensus constraints in the clique induced by $b_i$. EdgeCount can be computed based on the sum of $V_i$ over all Boolean variables, i.e., $\sum_{b_i \in \mathbb{B}} V_i$. As having multiple inconsistent Boolean variables on one edge is only counted once in EdgeCount, $\sum_{b_i \in \mathbb{B}} V_i$ overestimates EdgeCount. In the worst case, every violated consensus constraint is counted exactly $\sigma$ times in $\sum_{b_i \in \mathbb{B}} V_i$. Therefore, $\frac{1}{\sigma}\sum_{b_i \in \mathbb{B}} V_i$ gives a lower bound for EdgeCount.

We now consider the maximum for $V_i$ in any candidate solution. In other words, we want to assign each vertex either true or false in the clique of size $\kappa_i$, such that there are as many edges that connects two vertices with different labels as possible. This is essentially a graph vertex coloring problem [178]. A clique of size $\kappa_i$ is $\kappa_i$-colorable, which means the maximum for $V_i$ can never be as large as $\binom{\kappa_i}{2}$ for $\kappa_i > 2$. Therefore, we need to remove as few edges as possible (corresponding to satisfying consensus constraints) in the clique to make the remaining graph becomes 2-colorable.

In fact, the 2-colorable graphs are exactly the bipartite graphs. Our problem now reduces to finding a complete bipartite graph on $\kappa_i$ vertices, such that number of edges in the bipartite graph is maximal (an edge in the bipartite graph represents a violated consensus constraint). Let $p_i$ (number of positive votes) be the number of vertices on one side of the complete bipartite graph, and $n_i$ (number of negative votes) be the number of vertices on the other side of the complete bipartite graph. The number of edges in the complete bipartite is then $p_i \times n_i$, which is maximized when $n_i = p_i = \frac{\kappa_i}{2}$. The maximum for $V_i$ is then $\frac{\kappa_i^2}{4}$. The maximum for EdgeCount is at least $\frac{1}{\sigma}\sum_{b_i \in \mathbb{B}} V_i = \frac{1}{4\sigma}\sum_{b_i \in \mathbb{B}} \kappa_i^2$. Since the EdgeCount evaluation function can output any integer between 0 and the maximum $\frac{1}{4\sigma}\sum_{b_i \in \mathbb{B}} \kappa_i^2$, the cardinality for EdgeCount function is $\frac{1}{4\sigma}\sum_{b_i \in \mathbb{B}} \kappa_i^2$ (omitting the constant term 1). □

Comparing the results in Lemma 1 and Theorem 8, we can conclude that EdgeCount in MIG increases the cardinality of the evaluation than MinVote in CMCF, especially when $\kappa_i$ is large and $\sigma$ is small. Fortunately, both are true in our case. On one hand, equivalence constraint propagations (see Subsection 5.2.3) replace the all (negative) equivalent variables using a representative variable, resulting in increase $\kappa_i$ for the Representative Boolean variable. On the other hand, $\sigma$ is limited to 6 in the greedy partitioning method we used (see Subsubsection 5.2.2). In practice, we find that the number of Boolean variables shared across two blocks is mostly 1, seldom 2, and never more than 2. $\sigma$ is very small (for example, 2) in practice. Both the large $\kappa_i$ and the small $\sigma$ enhance the advantage of EdgeCount over MinVote in terms of cardinality.

**Local Search over MIG**. We now discuss how to perform local search over the refined representation MIG. First generate a random initial vector of minterms, one minterm from each block. With fixed probability $p$, a *greedy move* is perform: randomly choose an inconsistent edge $e$. Select the minterm greedily from both vertices adjacent to $e$ such that it yields the best evaluation. Or, with probability $(1-p)$, a *random force move* is performed: randomly choose an inconsistent Boolean variable $v$, and force all vertices containing $v$ to a randomly chosen Boolean value. We call this local search *Gforce*.

$M_1$

| $b_1$ | $b_2$ | $b_3$ |
|---|---|---|
| ~~0~~ | ~~0~~ | ~~0~~ |
| 1 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 1 | 0 |

$\Longleftrightarrow$ $M_2$

| $b_2$ | $b_3$ | $b_4$ |
|---|---|---|
| 1 | 0 | 0 |
| 0 | 1 | 1 |
| 0 | 1 | 0 |
| ~~1~~ | ~~1~~ | ~~1~~ |

**Figure 5.12:** An example illustrating arc consistency propagation. Each row represents a satisfying minterm to the block.

**Arc Consistency Propagation**. Modeling the interaction between minterms using MIG inspires us to introduce a third constraint propagation to MIG, *Arc Consistency Propagation*. The concept of arc consistency is borrowed from the Constraint Satisfaction Problem (CSP) community [179].

**Definition 8** (Arc Consistency)**.** *Let $\mathbb{S}_i$ (or $\mathbb{S}_j$) be set of all satisfying minterms to block $M_i$ (or $M_j$), an edge $(M_i, M_j)$ in MIG is arc consistent, if for each $s_\alpha \in \mathbb{S}_i$ there is some $s_\beta \in \mathbb{S}_j$ such that $(s_\alpha, s_\beta)$ satisfies the consensus constraint on edge $(M_i, M_j)$, and vice versa.*

Any minterm that is not arc-consistent can never yield a valid solution, and thus can be eliminated. Arc Consistency Propagation is the process of eliminating minterms that are not arc consistent, until the all edges in the MIG are arc consistent. Together with Constraint Propagation and Equivalency Constraint Propagation (see Subsection 5.2.3), the three constraint propagations are all repeated successively, until no more minterm can be eliminated.

In Figure 5.12, $M_1$ and $M_2$ share two Boolean variables $b_2$ and $b_3$. There is no corresponding minterm from $M_2$ to work with $\overline{b_1} \wedge \overline{b_2} \wedge \overline{b_3}$ from $M_1$ to satisfy the consensus constraint between $M_1$ and $M_2$. Therefore, no valid solution can possibly contain the minterm $\overline{b_1} \wedge \overline{b_2} \wedge \overline{b_3}$, and it can be eliminated from $M_1$. Similarly, $b_1 \wedge b_2 \wedge b_3$ can also be eliminated from $M_2$.

# Empirical Results

The local search operations used by Gforce is very similar to the ones in CMCF-LS, both following the framework of WalkSAT. In fact, the random move in Gforce and CMCF-LS are *identical*, i.e., forcing all blocks containing a randomly chosen Boolean variable $b_i$ to reach a consensus state on a random truth assignment. The greedy moves in Gforce and CMCF-LS, on the

other hand, are conceptually similar. Both select the best available minterm that yields the highest improvement in the evaluation function. The major difference between Gforce and CMCF-LS is therefore the definition of evaluation function. We have shown that the evaluation function in Gforce has a provably higher cardinality than the one in CMCF-LS. The question now becomes whether the increased cardinality will improve performance.

We perform empirical study to address the question. For a fair comparison we also introduce the arc consistency propagation to CMCF-LS. CMCF-LS and Gforce are then compared on semiprime factoring instances and parity learning instances in terms of success rate and the number of evaluations using the same 100 million evaluation cutoff.

**Effectiveness of Search**. Similar to the experimental protocol in Subsection 5.2.5, the success rates of the 50 runs are used for measuring the effectiveness of search. Figure 5.13 reports the empirical results. Thanks to the increased cardinality of evaluation function, Gforce vastly and consistently improves over the success rate of CMCF-LS. For instance, Gforce solves the largest semiprime factoring instance (factoring 3599) reliably in all 50 runs, whereas CMCF-LS can solve it in only 3 out of 50 runs. The advantage of Gforce over CMCF-LS becomes more pronounced as the instances get larger. On the 5 parity learning instances, we found that both CMCF-LS and Gforce can consistently solve the all instances in all 50 runs. We next look at the number of evaluations to compare the two SLS solvers to learn which one requires less computational resources to solve the instances.

**Number of Evaluated Candidate Solutions**. Figure 5.14 shows a more detailed picture on the improvement of Gforce of CMCF-LS on semiprime factoring instances. We observe that the median of the number of evaluations by Gforce stay well below CMCF-LS on most instances. The advantage of Gforce over CMCF-LS in terms of number of evaluations can be up to one order of magnitude.

Despite the fact that both CMCF-LS and Gforce can solve all 5 parry learning instances reliably in all 50 runs, Figure 5.15 reveals that Gforce still uses less median number of evaluations on 4 out of the 5 parity learning instances than CMCF-LS.

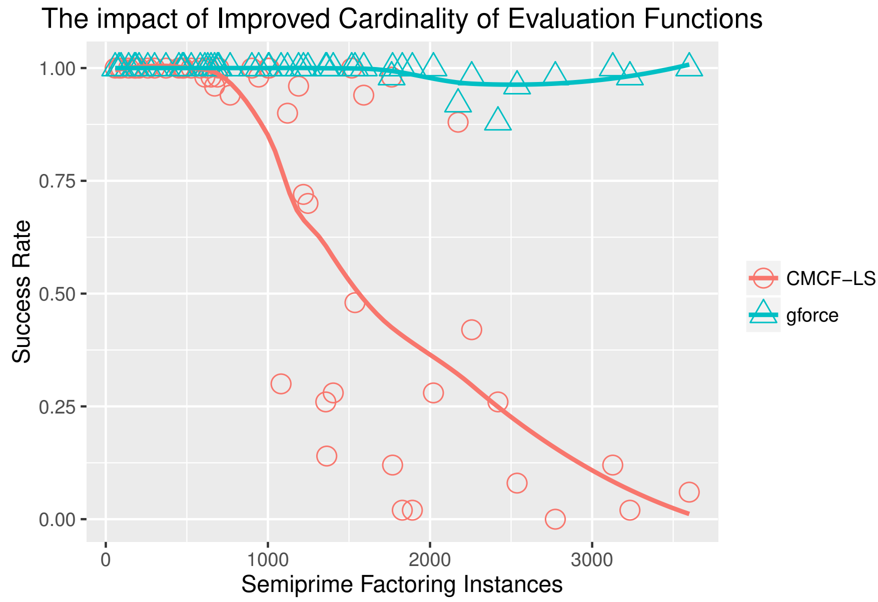



**Figure 5.13:** Success Rate on 50 Semiprime Factoring Problem Instances. The two trend curves are constructed using local polynomial regression fitting.

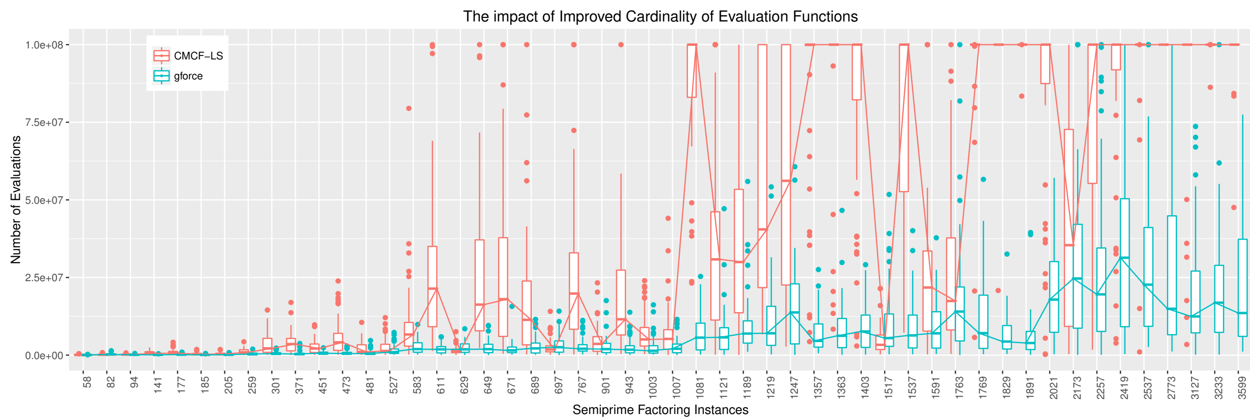



**Figure 5.14:** Number of Evaluations spent by CMCF-LS and Gforce on Semiprime Instances. Boxplot shows the distribution over 50 runs. Lines connect the medians.

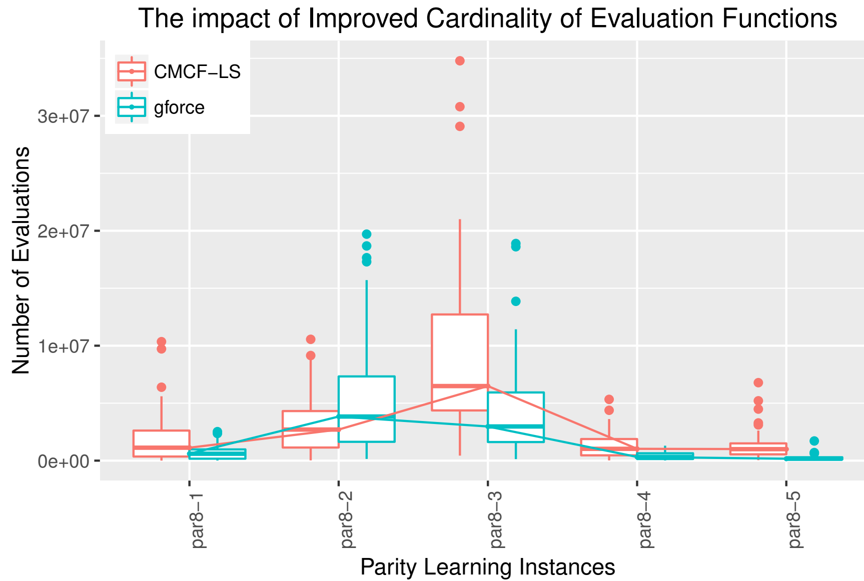



**Figure 5.15:** Number of Evaluations for CMCF-LS and Gforce on parity instances. Boxplot shows the distribution over 50 runs.

We have demonstrated that, by increasing the cardinality of evaluation function using MIG, Gforce shows significant improvement over CMCF-LS in terms of number of evaluations needed to solve an instance. In practice, however, the performance of a SAT solver is often measured by the raw CPU time. In the next subsection, we will study how to improve the raw CPU time, for the purpose of competing Gforce with the most advanced CNF-based SLS solvers.

### 5.3.2 Reducing Running Time with Partial Updates

## Theory

As SAT solvers in practice are usually measured by the raw CPU time needed to solve an instance, the best performing SLS solvers for SAT are usually highly optimized with respect to the running time. GSAT [14] is one of the first SLS SAT solvers; it has a significant impact on the design and implementation of a wide range of SAT solvers. GSAT always randomly select a variable that minimize the number of unsatisfied clauses. One key aspect of an efficient implementation of GSAT is caching and updating the variable score vector (i.e., the gradient with respect to current candidate solution under one-bit flip neighborhood) that form the basis for selecting the variable to

be flipped in each search step. A naive implementation requires scanning and evaluating all $\Theta(n)$ variables, in which every evaluation takes $\Theta(n)$ time. The time complexity for taking one move in a naive implementation is therefore $\Theta(n^2)$ (see Section 6.2 in [131]). Maintaining the variable score vector eliminates the need for scanning every variable for selecting the best move. Given the number of occurrence of any variable is bounded by a constant, (approximate) best improving moves can be identified in $\Theta(1)$ rather than $\Theta(n^2)$ in an naive implementation [132].

In the case of Gforce, maintaining a score vector is not immediately obvious, due to nature of MIG representation being rather different from the canonical CNF representation. As a first step, we introduce a straightforward optimization to Gforce, called *partial update*, to reduce the running time of evaluating candidate solutions.

Recall that the evaluation function in Gforce is the number of edges that violates the consensus constraint. A naive implementation iterates over all edges to check whether their corresponding consensus constraints are violated (see Algorithm 3). Let $E$ be the number of edges in a given MIG. Evaluating a candidate solution in the naive implementation takes $\Theta(E)$ time.

**Algorithm 3** A naive implementation of the evaluation function EdgeCount

```
1: Inputs: A MIG: G
           A candidate solution: S
2: Outputs: The EdgeCount evaluation of S: edgeCount
3: edgeCount ← 0
4: for e ∈ Edges(G) do
5:     if ViolateConsensusConstraint(e, S) then edgeCount ← edgeCount + 1
6: return edgeCount
```

Evaluating a candidate solution with *partial update* instead only checks the adjacent edges to the blocks that change their selected minterms. Algorithm 4 gives the pseudo code for implementing EdgeCount using partial update. The idea is that the new evaluation after changing a minterm are computed on the basis of the old evaluation, rather than computing the evaluation every time from scratch. Partial update requires the evaluation of the old candidate solution before the minterm changes. Partial update is achieved by adding “breaks” (constraints that were previously satisfied and

now violated, see line 3 of Algorithm 4) and subtracting "makes" (constraints that were previously unsatisfied and now satisfied, see line 6 of Algorithm 4). Calculating "makes" and "breaks" after each move requires keeping track of consensus constraint violation status for every edge. The Boolean vector $V$ is maintained for this purpose.

---

**Algorithm 4** An improved implementation of the evaluation function EdgeCount using partial update

---

1: **Inputs:** A MIG: $G$
 The EdgeCount evaluation of the old candidate solution before changing the selected minterm in block $m$: oldEdgeCount
 The new candidate solution after changing the selected minterm in $m$: **S**
 The vector of consensus constraint violation flags for every edge: $V$
2: **Outputs:** The EdgeCount evaluation of $S$
3: edgeCount ← oldEdgeCount
4: **for** $e \in$ IncidentEdges($m$, $G$) **do**
5: **if** ViolateConsensusConstraint($e$, **S**) **and** $V[e] = false$ **then**
6: edgeCount ← edgeCount + 1
7: $V[e] \leftarrow true$
8: **else if not** ViolateConsensusConstraint($e$, **S**) **and** $V[e] = true$ **then**
9: edgeCount ← edgeCount - 1
10: $V[e] \leftarrow false$
11: **return** edgeCount

---

Comparing Algorithm 4 with Algorithm 3 shows that the running time reduction with partial update is achieved by only checking the edges incident to the changed minterm. Lemma 2 rigorously addresses the correctness of this running time reduction technique.

**Lemma 2.** *After changing the selected minterm in a block $m$, only the edges incident to $m$ in the MIG can change their consensus constraint violation statuses.*

*Proof.* Proof by contradiction. Assume $e'$ is an edge that is not incident to $m$ and changes its consensus constraint violation status after $m$ selects a different minterm. Since $e'$ changes its constraint violation status, at least one of two blocks incident to $e'$ must have changed their selected minterm. As $m$ is the only block among all blocks that changes its selected minterm, $m$ is incident to $e'$. This is a contradiction to our assumption. □

We now study how focusing on checking the edges incident to the changed minterm, as in line 2 of Algorithm 4, gives rise to an asymptotic improvement in time complexity for evaluating a candidate solution. As the number of incident edges to a block can vary depending on the connectivity of the MIG, we conduct an amortized analysis over several iterations of Gforce under a mild assumption.

**Theorem 8.** *Let $\beta$ be number of blocks in a given MIG $G$ with $E$ edges, and $c$ be a constant, where $1 < c < \beta$. Suppose in a sequence of $\mu$ ($\mu > \beta$) iterations, each block is changed at most $\kappa$ times, where $\kappa$ be a constant that satisfies $\dfrac{\mu}{\beta} < \kappa < c\dfrac{\mu}{\beta}$. The amortized number of checked edges over $\mu$ iterations using partial update is $O(\frac{E}{\beta})$.*

*Proof.* Let us first consider a simpler case, where each minterm changes exactly once in a sequence of $\beta$ iterations, the number of checked edges using partial update is

$$\sum_{m \in \mathbb{M}} deg(m) = 2E, \tag{5.3}$$

where $deg(m)$ is the degree of vertex (block) $m$ in $G$.

Now consider the extreme case where every block changes exactly $\kappa$ times. This requires more than $\mu$ moves (i.e., $\kappa\beta > \mu$), because there are $\beta$ blocks. The extreme case thus gives an *upper bound* on the running time cost of $\mu$ iterations. The number of checked edges in the extreme case is

$$\sum_{m \in \mathbb{M}} \kappa \times deg(m) = 2\kappa E < 2c\frac{\mu}{\beta}E. \tag{5.4}$$

Amortizing the cost over $\mu$ evaluations, the number of checked edges in the extreme case is $O(\dfrac{2c\mu E}{\beta} \times \dfrac{1}{\mu}) = O(\dfrac{2cE}{\beta}) = O(\dfrac{E}{\beta})$. □

Recall that the naive implementation requires checking $\Theta(E)$ edges. Using partial update, which checks $O(\frac{E}{\beta})$, clearly yields an asymptotic improvement over the naive implementation. We next demonstrate via empirical study how the asymptotic improvement turns into practical performance in terms of CPU time.

## Empirical Results

In previous empirical study, we only compare the SLS solvers based on the alternative representations against the base solver WalkSAT. As the search operators in CMCF-LS and Gforce are both based on the framework of WalkSAT, the empirical comparison highlights the utility of the alternative representations, i.e., effectively reducing the number of evaluated candidate solutions to solve an application instance. Also recall that CMCF-LS in its naive implementation was unstable to match the raw performance of the highly optimized WalkSAT in terms of CPU time (see Subsection 5.2.5). Since the introduction of the partial update technique, we evaluate Gforce on semiprime factoring instances, to study how Gforce equipped with a straightforward optimization technique stands against the highly optimized CNF-based WalkSAT, in terms of CPU time. Moreover, we also compare the mildly optimized Gforce against three best performing SLS solvers on application SAT instances: SAPS [39], AdaptG$^2$WSAT [16] and Sparrow [40].

SAPS has been considered as one of the best performing local search solver on structured benchmarks [152]. AdaptG$^2$WSAT won several medals in SAT competitions. It won silver and bronze medals in 2007 SAT competition, and bronze medal in 2009 SAT competition. Sparrow won the Random SAT track from the SAT Competition 2011, and was also the best performing SLS in the crafted SAT track and application SAT track of SAT competition 2014. We use the UBCSAT (version v1.2beta18) [17] implementation, the most efficient implementation that we are aware of, in our empirical study.

As the SLS solvers in this study are much more powerful now, we generate 27 larger semiprime factoring instances. Before going into the details of how the 27 instances are selected, we need to address the relationship between semiprime to factor and the number of Boolean variables needed to encode the semiprime factoring problem in CNF representation. Figure 5.16 show the relationship for all semiprimes up to 65509. It appears like a step function; we only need more variables to encode the problem when the semiprime surpasses a certain point. There are only 9 discrete levels for number of Boolean variables, which divides the semiprimes into 9 groups. Assuming the number

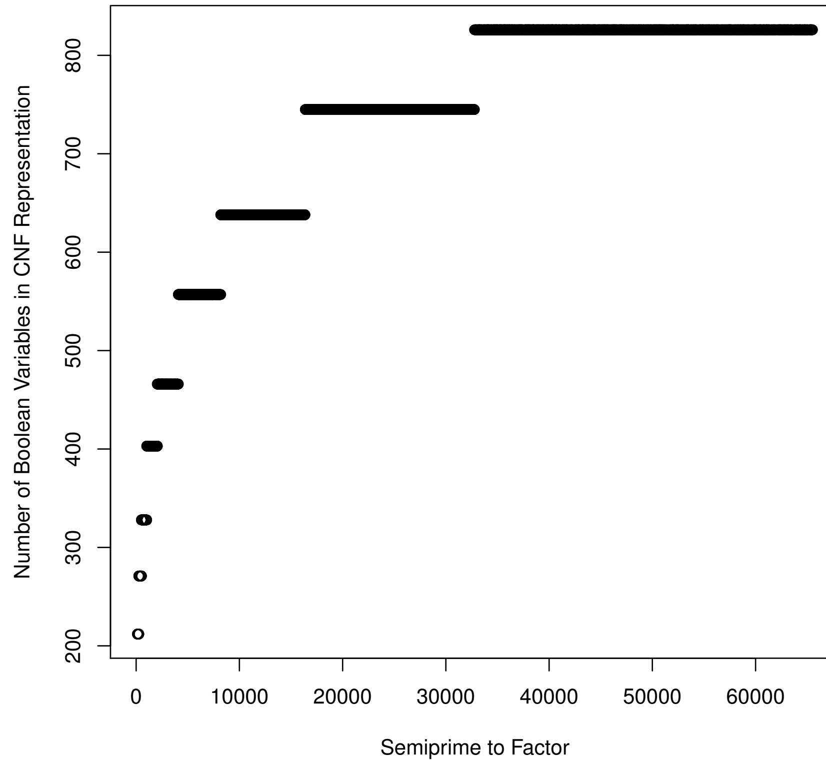



**Figure 5.16:** Relationship between semiprime to factor and the number of Boolean variables needed to encode the semiprime factoring problem in the CNF representation.

of Boolean variables indicates the difficulty of the problem, we randomly sample 3 instances from each of the 9 groups, resulting in 27 instances.

We run the 4 CNF-based SLS solvers and Gforce on the 27 instances for 10 runs, each run is limited to one hour, and record the CPU time spent by each solver to solve the instance. The success rate out of the 10 runs for each solver is reported in Figure 5.17. Using the simple framework of WalkSAT and the straightforward optimization, Gforce outperforms 3 highly efficient SLS solvers (WalkSAT, SAPS and AdaptG$^2$WSAT ) on the semiprime factoring, and matches the performance of the recent winner of SAT competitions (Sparrow). Gforce and Sparrow consistently solve all 27 instances in every of the 10 runs within the one hour limit. WalkSAT, which SLS framework is the basis for Gforce, already cannot reliably solve the instance of semiprime 1673 in all 10 runs, and fail to solve instance of any semiprime larger than 5951 in any of the 10 runs. Gforce demonstrates very

good capability at solving semiprime factoring problems, which rivals the recent SAT competition winner Sparrow, despite its simple form and straightforward optimization.

Now that both Gforce and Sparrow can both reliably all 27 factoring instances, we next study which one solves them faster. Figure 5.18 reports a comparison on the CPU time spent by Gforce and the 4 CNF-based rivals. The CPU time distribution boxplots demonstrate an even more pronounced lead of Gforce over its based solver WalkSAT. WalkSAT starts to fail to solve the instance factoring semiprime 1673 with the one hour limit, while the median CPU time for Gforce to solve the same instance is just 0.49 seconds. The instance factoring 5951 is the first one that WalkSAT cannot solve in any of the 10 runs, while Gforce reliably solves the same instance with median CPU time of 7.06 seconds. Compared with SAPS and AdaptG$^2$WSAT , Gforce not only solves more instances, but also solves the instances faster. On instances where both SAPS and AdaptG$^2$WSAT have a median solving time lower than the one hour limit, the speedup of Gforce in terms of median solving time is up to $129\times$ over AdaptG$^2$WSAT , and up to $21\times$ over SAPS. Finally, Gforce is neck and neck with the recent competition winner Sparrow. In term of median solving time, Gforce is faster than Sparrow on 13 out of 27 instances. What is encouraging is that Gforce is actually $6.17\times$ faster than Sparrow on the instance factoring the largest semiprime in our experiment 44003.

The empirical results have shown that, despite the fact that the time complexity per evaluation $O(\frac{E}{\beta})$ is not as good as $O(1)$ in many highly optimized CNF-based solver and that the current local search frame in Gforce is just as straightforward as in WalkSAT, Gforce manages to beat its base solver WalkSAT, two good performing SLS solvers on application instances SAPS and AdaptG$^2$WSAT , and competes well with the winner of several recent SAT competitions Sparrow. The encouraging results highlight the utility of the alternative representation MIG for improving the performance of the SLS solvers on application instances, and also leave considerable room for further improvement.

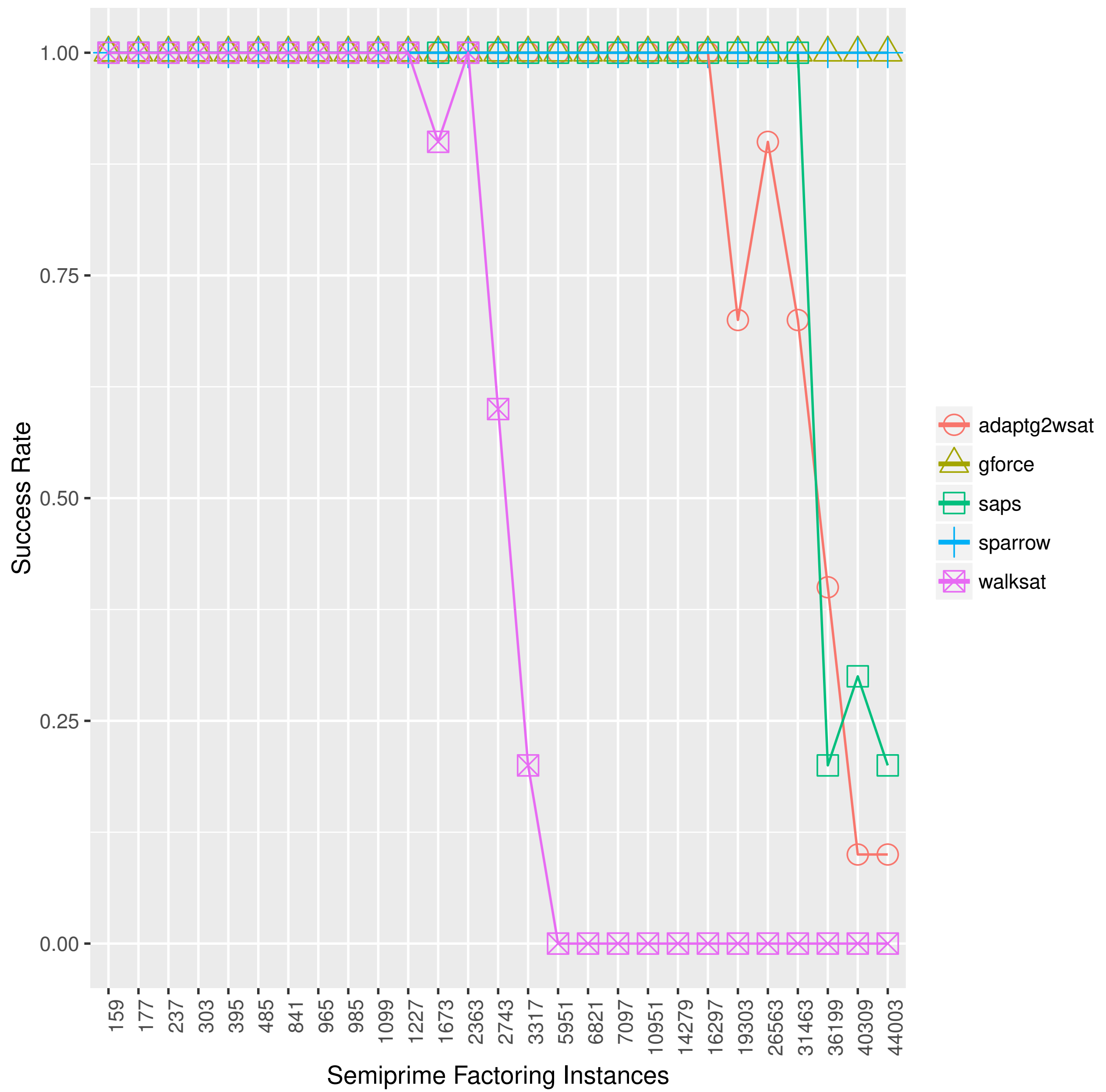



**Figure 5.17:** Success rate of on 27 semiprime factoring instances.

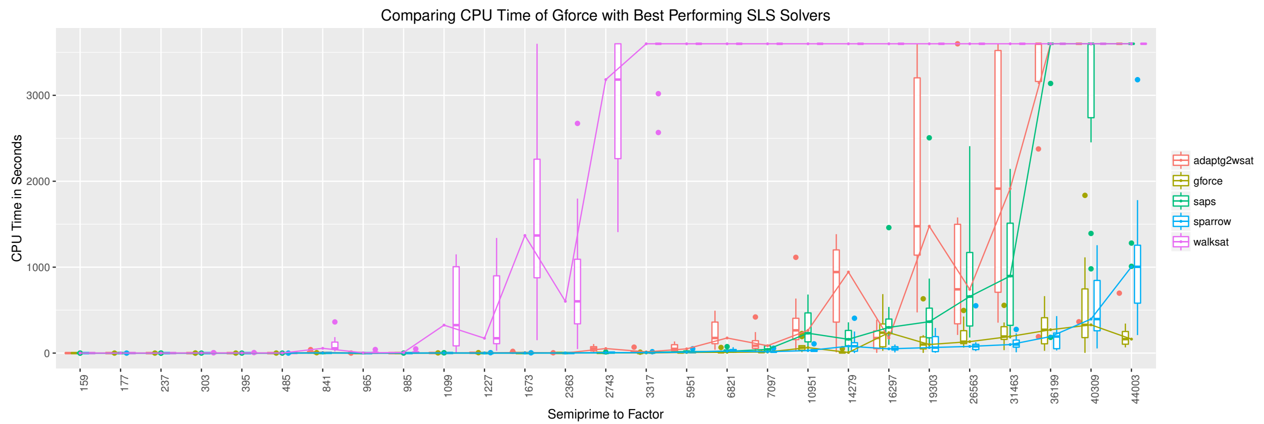



**Figure 5.18:** CPU time in seconds of Gforce and best performing SLS solvers on 27 semiprime factoring instances. Boxplot shows the distribution over 10 of one hour run.

# Chapter 6

# Conclusion and Future Work

Systematic Search (SS) solvers have been exploiting various structural properties with success on application instances. Stochastic Local Search (SLS) solvers, on the hand, drop their performance drastically on application instances. We hypothesize that SLS solvers can be improved by leveraging structural properties on application instances. We focus on two structural properties of SAT instances in this work: *variable interaction topology* and *subproblem constrainedness*. By analyzing and exploiting the two structural properties, we have successfully improve upon the state-of-art SLS solvers on a diverse set of application instances. Despite the promising results we have achieved by exploiting structural properties, there are various ways the current line of work can be further improved.

## 6.1 Future Work

### 6.1.1 Partition Crossover for SLS SAT Solvers

Our promising pilot work in Chapter 4 on improving SLS SAT solvers with Partition Crossover (PX) opens up many interesting future directions that requires further investigation. PX takes linear time to decompose the VIGs, as it is currently employing a top-down approach for decomposition. The current PX starts from the original VIG and removes the vertices whose assignment is shared among local optima. As an alternative, we can construct the decomposition in a bottom-up manner by tracking how the second parent deviates from the first parent, and incrementally add vertices to the recombination graph. This could potentially result in a reduction of the time complexity from $O(n)$ to $O(1)$.

In addition, we have learned from our empirical studies that there is a bell-shape relationship between $\alpha$ and the number of components. Finding the optimal setting of $\alpha$ can be achieved by performing golden section search that successively narrows the range containing the optimal $\alpha$.

### 6.1.2 Local Search over Minterm Interaction Graphs

As shown in Chapter 5, Minterm Interaction Graph (MIG) is a promising presentation alternative to the canonical CNF representation in improving the performance of SLS solvers on application SAT instances. We also identify several ways to further explore the utility of MIG.

## Comparison with Preprocessors

Using minterms as the infrastructure allows multiple straightforward and yet effective constraint propagations for reducing search space. Although the constraint propagations techniques are natural components of Gforce, they can also be viewed as a preprocessor since they are only invoked once upfront. Preprocessors also exist for CNF representation that usually involve unit propagation and resolution. It would be interesting to address the following questions:

1. Is the advantage of Gforce over WalkSAT a result of the use of the preprocessor in Gforce?

2. Can the minterm-based preprocessor complement the CNF-based preprocessor to further reduce the search space?

The first question can be answered by incorporating CNF-based preprocessors into WalkSAT. One can test two wide-used CNF-based preprocessors: SatElite [102] and Coprocessor [153]. One can compare the Gforce and WalkSAT equipped with the preprocessors in terms of success rate and CPU time on the semiprime factoring instances and parity learning instances. We expect to understand how much the preprocessors can lift the performance of WalkSAT and whether WalkSAT can beat Gforce with the help of the CNF-based preprocessors.

The second question can be addressed by running both the minterm-based preprocessor and CNF-based preprocessors and comparing the result with the one using only CNF-based preprocessors. One can also investigate which preprocessor should be invoked first. The utility of the hybrid preprocessor can be evaluated in two metrics: the size of the formula after preprocessing, and the total running time of the solver including the preprocessing phase.

## Transferring Advanced Techniques in CNF-based SLS Solvers

Despite the simple local search framework inherited from WalkSAT and the straightforward optimization with partial update, Gforce already outperforms several highly optimized SLS solvers and rivals the best performing CNF-based SLS solver Sparrow, thanks to the power of the alternative representation MIG. Meanwhile, Li and Huang [164] point out that the greedy move in WalkSAT can possibly select an unsatisfied clause that only contains disimproving move, even though improve moves are available in other clauses. $G^2$WSAT [164] enhances the greedy move of WalkSAT by employing a gradient-based approach that guarantees to find all improving moves efficiently. In fact, Whitley, Howe and Hains show that approximate best improving move and first improving move can be identified in constant time [132]. Empirical results confirm that $G^2$WSAT is substantially better than WalkSAT on a wide variety of problem instances.

The limitation of the greedy move in WalkSAT still remains in the MIG representation; Gforce can possibly select an edge whose incident vertices containing no improving minterm. We can introduce the gradient-based greedy move from $G^2$WSAT to Gforce to unleash more potential of the MIG representation.

Implementing the gradient-based greedy move in MIG is non-trivial. Let $n$ be the number of Boolean variables in a given instance. The neighborhood in CNF representation is simply all $n$ variables. The gradient-based method allows efficient identification of improving moves among the $n$ neighbors. It is achieved by maintaining a *score vector* of size $n$, in which every element stores the change in evaluation function upon moving towards a neighbor. In case of MIG representation, the neighborhood has a two-level structure: first selecting a block, and then selecting a minterm from that block. We can flatten two-level structure. Let $\mathbb{S}_i$ be set of all satisfying minterms to a block $M_i$, there are ($|\mathbb{S}_i|$ - 1) neighbors associated with $M_i$ (excluding the currently selected minterm). The size of total neighborhood (i.e., the length of score vector in MIG) is $\sum_{M_i \in P} (|\mathbb{S}_i| - 1) = \sum_{M_i \in P} |\mathbb{S}_i| - P$.

After introducing gradient-based greedy move, we expect Gforce to enhance its success on semiprime factoring instances, possibly outperforms the recent competition winner Sparrow. In fact, $G^2$WSAT is one of major components in Sparrow. Implementing gradient-based greedy move

in Gforce also sets a foundation for bringing more advanced techniques from Sparrow to the MIG representation.

## Better Partitioning

The future work in Subsection 6.1.2 is expected to enhance the success of Gforce in the current benchmarks, namely semiprime factoring instances and parity learning instances. We can potentially *extend* the success of MIG to more application instances.

We are currently using a simple greedy partitioning method (see Subsubsection 5.2.2). It is a reasonable starting point, given that we have seen natural blocking in several application instance classes. However, if the clauses belonging to one block are scattered across the instances, the greedy partitioning method will probably fail to group them into one block. To resolve this issue and to make the minterm-based alternative representation more applicable, we can formulate the partitioning of CNF clauses as a *clustering* problem. We define the distance between two CNF clauses as the number of *different variables*, i.e., $dist = (Vars(c_i) \setminus Vars(c_j)) \bigcup (Vars(c_j) \setminus Vars(c_i))$. For example, when two clauses involves the exact same set of variables, the distance between them is zero. We can then run adaptive k-means [180, 181] to partition the CNF clause set in a principled fashion. Even though exact k-means clustering is NP-hard [182], efficient heuristic algorithms exist. In practice, Lloyd's algorithm is considered to be of linear time complexity [183].

The new clustering formulation of the partitioning problem also allows removing the limitation that every block contains at most 6 variables. We have observed in several application instance classes (e.g., crypto-sha instances in the industrial track of 2014 SAT competition) that contain natural blocks with more than 6 variables. On such cases, the current greedy partition method will disrupt the natural blocks by breaking long clauses into smaller clauses such that each smaller clauses containing at most 6 variables. The clustering-based approach instead preserves the natural blocking structure and is capable of grouping the long clauses into the same block for minterm enumeration, as they all shared the same set of variables.

Finally, the clustering formulation leads to a hybrid representation that allows CNF clauses to co-exist with minterm-based blocks. Ansótegui, María and Levy reports that the CNF clause length in application instances typically follow a power law distribution, where a small number of exponentially long clauses exist [27]. In expectation, such exponentially long clauses are easier to satisfy; setting any literal to true among the many literals in the clauses is sufficient. As a result, it makes more sense to keep the long clauses as they are, rather than breaking them into fixed-length clauses and enumerating the numerous minterms associated with them. With hybrid representation, local search is performed in the minterm-based blocks only. Whenever a solution is found, it is checked against the CNF clauses. Only if the solution also satisfies the CNF long clauses, it is a valid solution to the original formula.

# Bibliography


[1] Carsten Sinz. Visualizing SAT instances and runs of the DPLL algorithm. *Journal of Automated Reasoning*, 39(2):219–243, 2007.

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

[3] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[4] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[5] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[6] Miroslav N. Velev and Randal E. Bryant. Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors. *J. Symb. Comput.*, 35(2):73–106, February 2003.

[7] Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[8] Boris Konev and Alexei Lisitsa. *A SAT Attack on the Erdős Discrepancy Conjecture*, pages 219–226. Springer International Publishing, Cham, 2014.

[9] Boris Konev and Alexei Lisitsa. Computer-aided proof of erdos discrepancy properties. *Artificial Intelligence*, 224:103 – 118, 2015.

[10] David S. Johnson and Michael A. Trick. Satisfiability Suggested Format. In *Second DIMACS Implementation Challenge*, 1992.

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

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

[13] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer Berlin Heidelberg, 2004.

[14] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, AAAI'92, pages 440–446. AAAI Press, 1992.

[15] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (vol. 1)*, AAAI '94, pages 337–343, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence.

[16] Chu Min Li, Wanxia Wei, and Harry Zhang. Combining adaptive noise and look-ahead in local search for SAT. In João Marques-Silva and Karem A. Sakallah, editors, *The 10th International Conference on Theory and Applications of Satisfiability Testing*, volume 4501 of *Lecture Notes in Computer Science*, pages 121–133. Springer, 2007.

[17] Dave A. D. Tompkins and Holger H. Hoos. UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT. In *Revised Selected Papers from the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, pages 306–320, 2005.

[18] Shaowei Cai and Kaile Su. Configuration checking with aspiration in local search for sat, 2012.

[19] Hans van Maaren and John Franco. The International SAT Competitions Webpage. http://www.satcompetition.org/.

[20] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the Really Hard Problems Are. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, IJCAI-91, Sydney, Australia*, pages 331–337, 1991.

[21] Ian P Gent and Toby Walsh. The sat phase transition. In *ECAI*, volume 94, pages 105–109. PITMAN, 1994.

[22] Remi Monasson, Riccardo Zecchina, Scott Kirkpatrick, Bart Selman, and Lidror Troyansky. Determining computational complexity from characteristic 'phase transitions'. *Nature*, 400(6740):133–137, July 1999.

[23] Henry Yuen and Joseph Bebel. ToughSAT Generation. https://toughsat.appspot.com/, 2011. Accessed: April 11th, 2016.

[24] Norbert Manthey. Generating clique coloring problem formulas. In Anton Belov, Daniel Diepold, Marijn J.H. Heule, and Matti Järvisalo, editors, *Proceedings of SAT Competition 2014*, volume B-2014-2 of *Department of Computer Science Series of Publications B*, page 89. University of Helsinki, Helsinki, Finland, 2014.

[25] Chu-Min Li and Bing Ye. Sat-encoding of step-reduced md5. In Anton Belov, Daniel Diepold, Marijn J.H. Heule, and Matti Järvisalo, editors, *Proceedings of SAT Competition 2014*, volume B-2014-2 of *Department of Computer Science Series of Publications B*, page 94. University of Helsinki, Helsinki, Finland, 2014.

[26] Carla Gomes and Toby Walsh. Randomness and structure. *Handbook of Constraint Programming*, pages 639–664, 2006.

[27] Carlos Ansótegui, María Luisa Bonet, and Jordi Levy. On the structure of industrial sat instances. In *International Conference on Principles and Practice of Constraint Programming*, pages 127–141. Springer, 2009.

[28] Ignasi Abío Roig et al. *Solving hard industrial combinatorial problems with SAT*. PhD thesis, Universitat Politècnica de Catalunya, 2013.

[29] Eyal Amir and Sheila McIlraith. Partition-based logical reasoning for first-order and propositional theories. *Artificial intelligence*, 162(1):49–88, 2005.

[30] Vijay Durairaj and Priyank Kalla. Guiding cnf-sat search via efficient constraint partitioning. In *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 498–501. IEEE Computer Society, 2004.

[31] Alan M Frisch, Timothy J Peugniez, Anthony J Doggett, and Peter W Nightingale. Solving non-boolean satisfiability problems with stochastic local search: A comparison of encodings. *Journal of Automated Reasoning*, 35(1-3):143, 2005.

[32] Steven David Prestwich. CNF Encodings. *Handbook of Satisfiability*, 185:75–97, 2009.

[33] Bart Selman, Henry Kautz, and David McAllester. Ten challenges in propositional reasoning and search. In *Proceedings of the 15th International Joint Conference on Artifical Intelligence - Volume 1*, IJCAI'97, pages 50–54, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[34] Henry Kautz and Bart Selman. Ten challenges redux: Recent progress in propositional reasoning and search. In *Principles and Practice of Constraint Programming–CP 2003*, pages 1–18. Springer, 2003.

[35] Zack Newsham, William Lindsay, Vijay Ganesh, Jia Hui Liang, Sebastian Fischmeister, and Krzysztof Czarnecki. Satgraf: Visualizing the evolution of sat formula structure in solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 62–70. Springer, 2015.

[36] Eyal Amir and Sheila Mcilraith. Solving satisfiability using decomposition and the most constrained subproblem. In *in LICS workshop on Theory and Applications of Satisfiability Testing (SAT*, 2001.

[37] Wenxiang Chen and Darrell Whitley. Decomposing sat instances with pseudo backbones. In Bin Hu and Manuel López-Ibáñez, editors, *Evolutionary Computation in Combinatorial Optimization: 17th European Conference, EvoCOP 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings*, pages 75–90, Cham, 2017. Springer International Publishing.

[38] Wenxiang Chen, L. Darrell Whitley, Adele E. Howe, and Brian Goldman. Stochastic local search over minterms on structured SAT instances. In Jorge A. Baier and Adi Botea, editors, *Proceedings of the Ninth Annual Symposium on Combinatorial Search, SOCS 2016, Tarrytown, NY, USA, July 6-8, 2016.*, pages 125–126. AAAI Press, 2016.

[39] Frank Hutter, Dave A. D. Tompkins, and Holger H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for sat. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming*, CP '02, pages 233–248, London, UK, UK, 2002. Springer-Verlag.

[40] Adrian Balint and Andreas Fröhlich. Improving stochastic local search for sat with a new probability distribution. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 10–15. Springer, 2010.

[41] Marc Mézard, Giorgio Parisi, and Riccardo Zecchina. Analytic and algorithmic solution of random satisfiability problems. *Science*, 297(5582):812–815, 2002.

[42] David Mitchell, Bart Selman, and Hector Levesque. Hard and easy distributions of sat problems. In *AAAI*, volume 92, pages 459–465, 1992.

[43] Philip Kilby, John Slaney, Sylvie Thiébaux, Toby Walsh, et al. Backbones and backdoors in satisfiability. In *AAAI*, volume 5, pages 1368–1373, 2005.

[44] Olivier Dubois and Gilles Dequen. A backbone-search heuristic for efficient solving of hard 3-sat formulae. In *IJCAI*, volume 1, pages 248–253, 2001.

[45] Eric I. Hsu, Christian J. Muise, J. Christopher Beck, and Sheila A. McIlraith. *Probabilistically Estimating Backbones and Variable Bias: Experimental Overview*, pages 613–617. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[46] Eric I Hsu and Sheila A McIlraith. Varsat: Integrating novel probabilistic inference techniques with dpll search. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 377–390. Springer, 2009.

[47] Weixiong Zhang, Ananda Rangan, and Moshe Looks. Backbone guided local search for maximum satisfiability. In Georg Gottlob and Toby Walsh, editors, *Proc. International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1179–1186. Morgan Kaufmann, 2003.

[48] Weixiong Zhang. Configuration landscape analysis and backbone guided local search: part i: Satisfiability and maximum satisfiability. *Artificial Intelligence*, 158:1–26, September 2004.

[49] Mikoláš Janota, Inês Lynce, and Joao Marques-Silva. Algorithms for computing backbones of propositional formulae. *AI Communications*, 28(2):161–177, 2015.

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

[51] Toby Walsh and John Slaney. Backbones in optimization and approximation. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01)*, 2001.

[52] Andrew J Parkes. Clustering at the phase transition. In *AAAI/IAAI*, pages 340–345. Citeseer, 1997.

[53] Josh Singer, Ian P. Gent, and Alan Smaill. Backbone fragility and the local search cost peak. *Journal of Artificial Intelligence Research (JAIR)*, 12:235–270, 2000.

[54] Chu Min Li and Anbulagan Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the 15th international joint conference on Artifical intelligence-Volume 1*, pages 366–371. Morgan Kaufmann Publishers Inc., 1997.

[55] Gilles Dequen and Olivier Dubois. Kcnfs: An efficient solver for random k-sat formulae. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 486–501. Springer, 2003.

[56] Ryan Williams, Carla P Gomes, and Bart Selman. Backdoors to typical case complexity. In *IJCAI*, volume 3, pages 1173–1178. Citeseer, 2003.

[57] Stefan Szeider. Backdoor sets for dll subsolvers. *Journal of Automated Reasoning*, 35(1-3):73–88, 2005.

[58] Lionel Paris, Richard Ostrowski, Pierre Siegel, and Lakhdar Sais. Computing horn strong backdoor sets thanks to local search. In *2006 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'06)*, pages 139–143. IEEE, 2006.

[59] Zijie Li and Peter Van Beek. Finding small backdoors in sat instances. In *Canadian Conference on Artificial Intelligence*, pages 269–280. Springer, 2011.

[60] Naomi Nishimura, Prabhakar Ragde, and Stefan Szeider. Detecting backdoor sets with respect to horn and binary clauses. *SAT*, 4:96–103, 2004.

[61] Bistra Dilkina, Carla P Gomes, and Ashish Sabharwal. Tradeoffs in the complexity of backdoor detection. In *International Conference on Principles and Practice of Constraint Programming*, pages 256–270. Springer, 2007.

[62] Carla P Gomes, Bart Selman, Henry Kautz, et al. Boosting combinatorial search through randomization. *AAAI/IAAI*, 98:431–437, 1998.

[63] Bart Selman Ryan Williams, Carla Gomes. On the connections between heavy-tails, backdoors, and restarts in combinatorial search. In *SAT*, 2003.

[64] Yannet Interian. Backdoor sets for random 3-sat. In *SAT*, pages 231–238, 2003.

[65] Yongshao Ruan, Henry A Kautz, and Eric Horvitz. The backdoor key: A path to understanding problem hardness. In *AAAI*, volume 4, pages 118–123, 2004.

[66] Carla P. Gomes, Bart Selman, Nuno Crato, and Henry A. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24(1/2):67–100, 2000.

[67] Stephan Kottler, Michael Kaufmann, and Carsten Sinz. A new bound for an np-hard subclass of 3-sat using backdoors. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 161–167. Springer, 2008.

[68] Marko Samer and Stefan Szeider. Backdoor trees. In *AAAI*, volume 8, pages 13–17, 2008.

[69] Serge Gaspers and Stefan Szeider. Backdoors to satisfaction. In *The Multivariate Algorithmic Revolution and Beyond*, pages 287–317. Springer, 2012.

[70] Serge Gaspers, Neeldhara Misra, Sebastian Ordyniak, Stefan Szeider, and Stanislav Živny. Backdoors into heterogeneous classes of sat and csp. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 2652–2658. AAAI Press, 2014.

[71] Steven David Prestwich. Variable dependency in local search: Prevention is better than cure. In João Marques-Silva and Karem A. Sakallah, editors, *Proceesing of The International Conferences on Theory and Applications of Satisfiability Testing*, volume 4501 of *Lecture Notes in Computer Science*, pages 107–120. Springer, 2007.

[72] Henry Kautz, David McAllester, and Bart Selman. Exploiting variable dependency in local search. In *Proceeding of Poster Sessions of the Fifteenth International Joint Conference on Artificial Intelligence*, 1997.

[73] G.S. Tseitin. On the complexity of derivation in propositional calculus. In JörgH. Siekmann and Graham Wrightson, editors, *Automation of Reasoning*, Symbolic Computation, pages 466–483. Springer Berlin Heidelberg, 1983.

[74] Joao Marques-Silva. *Search algorithms for satisfiability problems in combinational switching circuits*. PhD thesis, University of Michigan, 1995.

[75] Christos H Papadimitriou. On selecting a satisfying truth assignment. In *Proceedings of 32nd Annual Symposium on Foundations of Computer Science*, pages 163–169. IEEE, 1991.

[76] Henry Kautz and Bart Selman. The state of SAT. *Discrete Applied Mathematics*, 155(12):1514 – 1524, 2007.

[77] Jérome Lang and Pierre Marquis. Complexity results for independence and definability in propositional logic. *KR*, 98:356–367, 1998.

[78] James M Crawford and Larry D Auton. Experimental results on the crossover point in random 3-sat. *Artificial intelligence*, 81(1):31–57, 1996.

[79] Duc Nghia Pham, John Thornton, and Abdul Sattar. Building Structure into Local Search for SAT. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, volume 7, pages 2359–2364, 2007.

[80] Duc Nghia Pham, John Thornton, and Abdul Sattar. Efficiently Exploiting Dependencies in Local Search for SAT. In *Proceedings of the Twenty-Third Conference on Artificial Intelligence*, pages 1476–1478, 2008.

[81] Holger H. Hoos. An adaptive noise mechanism for WalkSAT. In Rina Dechter and Richard S. Sutton, editors, *AAAI/IAAI*, pages 655–660. AAAI Press / The MIT Press, 2002.

[82] Richard Ostrowski, Éric Grégoire, Bertrand Mazure, and Lakhdar Sais. Recovering and exploiting structural knowledge from cnf formulas. In *International Conference on Principles and Practice of Constraint Programming*, pages 185–199. Springer, 2002.

[83] James M Crawford, Michael J Kearns, and RE Shapire. The minimal disagreement parity problem as a hard satisfiability problem. *Computational Intell. Research Lab and AT&T Bell Labs Technical Report*, 1994.

[84] Fahiem Bacchus and Jonathan Winter. Effective preprocessing with hyper-resolution and equality reduction. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and*

*Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 341–355. Springer Berlin Heidelberg, 2004.

[85] Éric Grégoire, Richard Ostrowski, Bertrand Mazure, and Lakhdar Saïs. Automatic extraction of functional dependencies. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 122–132. Springer, 2004.

[86] Bengt Aspvall, Michael F Plass, and Robert Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979.

[87] Irina Rish and Rina Dechter. Resolution versus search: Two strategies for sat. *Journal of Automated Reasoning*, 24(1-2):225–275, 2000.

[88] Renato Tinós, Darrell Whitley, and Francisco Chicano. Partition crossover for pseudo-boolean optimization. In *Proceedings of the 2015 ACM Conference on Foundations of Genetic Algorithms XIII*, pages 137–149. ACM, 2015.

[89] Carsten Sinz. Visualizing the internal structure of SAT instances (preliminary report). In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*, 2004.

[90] Carsten Sinz and Edda-Maria Dieringer. Dpvis–a tool to visualize the structure of sat instances. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 257–268. Springer, 2005.

[91] Thomas MJ Fruchterman and Edward M Reingold. Graph drawing by force-directed placement. *Software: Practice and experience*, 21(11):1129–1164, 1991.

[92] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world' networks. *nature*, 393(6684):440–442, 1998.

[93] Réka Albert and Albert-Lászlo Barabási. Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1):47, 2002.

[94] Toby Walsh. Search in a small world. In *IJCAI*, volume 99, pages 1172–1177, 1999.

[95] Toby Walsh. Search on high degree graphs. In *IJCAI*, volume 1, pages 266–274. Citeseer, 2001.

[96] Réka Albert, Hawoong Jeong, and Albert-László Barabási. Internet: Diameter of the world-wide web. *Nature*, 401(6749):130–131, 1999.

[97] Carlos Ansótegui, Jesús Giráldez-Cru, and Jordi Levy. The community structure of sat formulas. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 410–423. Springer, 2012.

[98] Armin Biere and Carsten Sinz. Decomposing SAT problems into connected components. *JSAT*, 2(1-4):201–208, 2006.

[99] Mark EJ Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.

[100] U. Brandes, D. Delling, M. Gaertler, R. Görke, M. Hoefer, Z. Nikoloski, and D. Wagner. On modularity - NP-Completeness and beyond. Technical Report 2006-19, Faculty of Informatics, Karlsruher Institut für Technologie (KIT), 2006.

[101] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E*, 76(3):036106, 2007.

[102] Niklas Eén and Armin Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing*, SAT'05, pages 61–75, Berlin, Heidelberg, 2005. Springer-Verlag.

[103] Armin Biere. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:75–97, 2008.

[104] Zack Newsham, Vijay Ganesh, Sebastian Fischmeister, Gilles Audemard, and Laurent Simon. Impact of community structure on sat solver performance. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 252–268. Springer, 2014.

[105] Nathan Mull, Daniel J Fremont, and Sanjit A Seshia. On the hardness of sat with community structure. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 141–159. Springer, 2016.

[106] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *IJCAI*, volume 9, pages 399–404, 2009.

[107] Carlos Ansótegui, Jesús Giráldez-Cru, Jordi Levy, and Laurent Simon. Using community structure to detect relevant learnt clauses. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 238–254. Springer, 2015.

[108] Robert Ganian and Stefan Szeider. Community structure inspired algorithms for sat and# sat. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 223–237. Springer, 2015.

[109] Jingchao Chen. A bit-encoding phase selection strategy for satisfiability solvers. In T. V. Gopal, Manindra Agrawal, Angsheng Li, and S. Barry Cooper, editors, *Theory and Applications of Models of Computation - 11th Annual Conference, TAMC 2014, Chennai, India, April 11-13, 2014. Proceedings*, volume 8402 of *Lecture Notes in Computer Science*, pages 158–167. Springer, 2014. minisat-blbd reference.

[110] Neil Robertson and Paul D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, 1986.

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

[112] Hans L Bodlaender. Dynamic programming on graphs with bounded treewidth. In *International Colloquium on Automata, Languages, and Programming*, pages 105–118. Springer, 1988.

[113] Sergios Theodoridis. *Machine learning: a Bayesian and optimization perspective*. Academic Press, 2015.

[114] Stefan Arnborg, Derek G Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987.

[115] Yu Wu, Per Austrin, Toniann Pitassi, and David Liu. Inapproximability of treewidth and related problems. *Journal of Artificial Intelligence Research*, 49:569–600, 2014.

[116] Hans L Bodlaender. Discovering treewidth. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 1–16. Springer, 2005.

[117] Vibhav Gogate and Rina Dechter. A complete anytime algorithm for treewidth. In *Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 201–208. AUAI Press, 2004.

[118] Fedor V Fomin, Dieter Kratsch, Ioan Todinca, and Yngve Villanger. Exact algorithms for treewidth and minimum fill-in. *SIAM Journal on Computing*, 38(3):1058–1079, 2008.

[119] Rong Zhou and Eric A Hansen. Combining breadth-first and depth-first strategies in searching for treewidth. In *IJCAI*, volume 9, pages 640–645, 2009.

[120] Hans L Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on computing*, 25(6):1305–1317, 1996.

[121] Robert Mateescu. Treewidth in industrial sat benchmarks. Technical report, Technical report, Microsoft Research, 2011.

[122] Armin Biere. Precosat system description. *SAT Competition, solver description*, 2009.

[123] Per Bjesse, James Kukula, Robert Damiano, Ted Stanion, and Yunshan Zhu. Guiding SAT diagnosis with tree decompositions. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 315–329. Springer, 2003.

[124] Adnan Darwiche and Mark Hopkins. Using recursive decomposition to construct elimination orders, jointrees, and dtrees. In *European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, pages 180–191. Springer, 2001.

[125] Jinbo Huang and Adnan Darwiche. A structure-based variable ordering heuristic for SAT. In *IJCAI*, volume 3, pages 1167–1172, 2003.

[126] We Li and Peter Van Beek. Guiding real-world sat solving with dynamic hypergraph separator decomposition. In *Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on*, pages 542–548. IEEE, 2004.

[127] Djamal Habet, Lionel Paris, and Cyril Terrioux. A tree decomposition based approach to solve structured sat instances. In *2009 21st IEEE International Conference on Tools with Artificial Intelligence*, pages 115–122. IEEE, 2009.

[128] Knot Pipatsrisawat and Adnan Darwiche. Rsat 2.0: Sat solver description. *SAT competition*, 7, 2007.

[129] Anthony Monnet and Roger Villemaire. Scalable formula decomposition for propositional satisfiability. In *Proceedings of the Third C* Conference on Computer Science and Software Engineering*, pages 43–52. ACM, 2010.

[130] Artan Dermaku, Tobias Ganzow, Georg Gottlob, Ben McMahan, Nysret Musliu, and Marko Samer. Heuristic methods for hypertree decomposition. In *Mexican International Conference on Artificial Intelligence*, pages 1–11. Springer, 2008.

[131] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search: Foundations & Application*. Morgan Kaufmann, 1 edition, September 2004.

[132] Darrell Whitley, Adele E. Howe, and Doug Hains. Greedy or not? best improving versus first improving stochastic local search for maxsat. In Marie desJardins and Michael L. Littman, editors, *AAAI*. AAAI Press, 2013.

[133] Doug Hains. *Structure in combinatorial optimization and its effect on heuristic performance*. PhD thesis, Colorado State University. Libraries, 2007.

[134] Jeremy D. Frank, Peter Cheeseman, and John Stutz. When gravity fails: Local search topology. *Journal of Artificial Intelligence Research*, 7:249–281, 1997.

[135] Darrell Whitley, Doug Hains, and Adele Howe. Tunneling between optima: partition crossover for the traveling salesman problem. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 915–922. ACM, 2009.

[136] Gabriela Ochoa, Francisco Chicano, Renato Tinós, and Darrell Whitley. Tunnelling crossover networks. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 449–456, New York, NY, USA, 2015. ACM.

[137] Shen Lin. Computer solutions of the traveling salesman problem. *The Bell System Technical Journal*, 44(10):2245–2269, 1965.

[138] Stuart Kauffman and Simon Levin. Towards a general theory of adaptive walks on rugged landscapes. *Journal of Theoretical Biology*, 128(1):11–45, 1987.

[139] Holger H Hoos. Sat-encodings, search space structure, and local search performance. In *IJCAI*, volume 99, pages 296–303. Citeseer, 1999.

[140] Edward Weinberger. Correlated and uncorrelated fitness landscapes and how to tell the difference. *Biological Cybernetics*, 63(5):325–336, September 1990.

[141] Terry Jones and Stephanie Forrest. Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In Larry J. Eshelman, editor, *International Conferecen on Genetic Algorithm (ICGA)*, pages 184–192. Morgan Kaufmann, 1995.

[142] Holger H. Hoos, Kevin Smyth, and Thomas Stützle. Search Space Features Underlying the Performance of Stochastic Local Search Algorithms for MAX-SAT. In Xin Yao, Edmund K. Burke, JoséA. Lozano, Jim Smith, JuanJulián Merelo-Guervós, JohnA. Bullinaria, JonathanE. Rowe, Peter Tiňo, Ata Kabán, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VIII*, volume 3242 of *Lecture Notes in Computer Science*, pages 51–60. Springer Berlin Heidelberg, 2004.

[143] Dale Schuurmans and Finnegan Southey. Local search characteristics of incomplete sat procedures. In Henry A. Kautz and Bruce W. Porter, editors, *AAAI/IAAI*, pages 297–302. AAAI Press / The MIT Press, 2000.

[144] Dale Schuurmans and Finnegan Southey. Local search characteristics of incomplete sat procedures. *Artificial Intelligence*, 132(2):121–150, 2001.

[145] Frank Harary et al. Graph theory, 1969.

[146] Zhe Wu and Benjamin W Wah. An efficient global-search strategy in discrete lagrangian methods for solving hard satisfiability problems. In *AAAI/IAAI*, pages 310–315, 2000.

[147] Holger H Hoos. On the run-time behaviour of stochastic local search algorithms for sat. In *AAAI/IAAI*, pages 661–666, 1999.

[148] Yuliya Zabiyaka and Adnan Darwiche. Functional treewidth: Bounding complexity in the presence of functional dependencies. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 116–129. Springer, 2006.

[149] Carlos Ansótegui, María Luisa Bonet, Jordi Levy, and Felip Manya. Measuring the hardness of sat instances. In *AAAI*, volume 8, pages 222–228, 2008.

[150] Serge Gaspers and Stefan Szeider. Strong backdoors to bounded treewidth SAT. In *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on*, pages 489–498. IEEE, 2013.

[151] Mohamed Qasem and Adam Prugel-Bennett. Learning the large-scale structure of the max-sat landscape using populations. *IEEE Transactions on Evolutionary Computation*, 14(4):518–529, 2010.

[152] Lukas Kroc, Ashish Sabharwal, Carla P. Gomes, and Bart Selman. Integrating systematic and local search paradigms: a new strategy for MaxSAT. In *Proceedings of the 21st international jont conference on Artifical intelligence*, IJCAI'09, pages 544–551, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[153] Norbert Manthey. Coprocessor 2.0–a flexible CNF simplifier. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 436–441. Springer, 2012.

[154] Wenxiang Chen and Darrell Whitley. Tunneling between local optima and plateaus on max-ksat using partition crossover. In *Submitted to the 32nd AAAI Conference on Artificial Intelligence*, 2018.

[155] Yibin Chen, Sean Safarpour, João Marques-Silva, and Andreas G. Veneris. Automated Design Debugging With Maximum Satisfiability. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 29(11):1804–1817, 2010.

[156] Chu-Min Li and Zhe Quan. An efficient branch-and-bound algorithm based on maxsat for the maximum clique problem. In *AAAI Conference on Artificial Intelligence*, 2010.

[157] Kerstin Bunte, Matti Järvisalo, Jeremias Berg, Petri Myllymäki, Jaakko Peltonen, and Samuel Kaski. Optimal neighborhood preserving visualization by maximum satisfiability. In *AAAI Conference on Artificial Intelligence*, pages 1694–1700. AAAI Press, 2014.

[158] Jeremias Berg, Matti Järvisalo, and Brandon Malone. Learning Optimal Bounded Treewidth Bayesian Networks via Maximum Satisfiability. In Samuel Kaski and Jukka Corander,

editors, *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*, volume 33 of *Proceedings of Machine Learning Research*, pages 86–95, Reykjavik, Iceland, 22–25 Apr 2014. PMLR.

[159] André Abramé and Djamal Habet. Ahmaxsat: Description and evaluation of a branch and bound max-sat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:89–128, 2015.

[160] Bart Selman and Henry A. Kautz. An empirical study of greedy local search for satisfiability testing. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, AAAI'93, pages 46–51. AAAI Press, 1993.

[161] Renato Tinós, Darrell Whitley, and Gabriela Ochoa. Generalized asymmetric partition crossover (GAPX) for the asymmetric TSP. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 501–508. ACM, 2014.

[162] Francisco Chicano, Darrell Whitley, Gabriela Ochoa, and Renato Tinós. Optimizing one million variable nk landscapes by hybridizing deterministic recombination and local search. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, pages 753–760, New York, NY, USA, 2017. ACM.

[163] Keld Helsgaun. An effective implementation of the Lin-Kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106 – 130, 2000.

[164] Chu Min Li and Wen Qi Huang. Diversification and determinism in local search for satisfiability. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing*, SAT'05, pages 158–172, Berlin, Heidelberg, 2005. Springer-Verlag.

[165] Ian P. Gent and Toby Walsh. An empirical analysis of search in gsat. *J. Artif. Int. Res.*, 1(1):47–59, September 1993.

[166] Monaldo Mastrolilli and Luca Maria Gambardella. Maximum satisfiability: How good are tabu search and plateau moves in the worst-case? *European Journal of Operational Research*,

166(1):63 – 76, 2005. Metaheuristics and Worst-Case Guarantee Algorithms: Relations, Provable Properties and Applications.

[167] Chuan Luo, Shaowei Cai, Wei Wu, Zhong Jie, and Kaile Su. CCLS: an efficient local search algorithm for weighted maximum satisfiability. *IEEE Trans. Computers*, 64(7):1830–1843, 2015.

[168] Henry B. Mann and Donald R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 18(1):50–60, 1947.

[169] J. Jensen. Sur les fonctions convexes et les inégalités entre les valeurs moyennes. *Acta Mathematica*, 30(1):175–193, December 1906.

[170] Darrell Whitley. Mk Landscapes, NK Landscapes, MAX-kSAT: A Proof That the Only Challenging Problems Are Deceptive. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 927–934, New York, NY, USA, 2015. ACM.

[171] Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems, Vol. I*. Computer Science Press, Inc., New York, NY, USA, 1988.

[172] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the state explosion problem in model checking. In *Informatics*, pages 176–194. Springer, 2001.

[173] Olivier Roussel. Another SAT to CSP conversion. In *Proceedings of 16th IEEE International Conference on Tools with Artificial Intelligence*, pages 558–565. IEEE, 2004.

[174] Frederick J. Hill and Gerald R. Peterson. *Introduction to Switching Theory and Logical Design*. John Wiley & Sons, Inc., New York, NY, USA, 3rd edition, 1981.

[175] João Carlos Leandro da Silva. Factoring semiprimes and possible implications for RSA. In *Proceedings of 2010 IEEE 26th Convention of Electrical and Electronics Engineers in Israel*, pages 000182–000183. IEEE, 2010.

[176] H. Hoos and T. Stutzle. SATLIB: An Online Resource for Research on SAT. In H. van Maaren I. P. Gent and T. Walsh, editors, *SAT2000*, pages 283–292. IOS Press, 2000.

[177] William S Cleveland, Eric Grosse, and William M Shyu. Local regression models. *Statistical models in S*, 2:309–376, 1992.

[178] Tommy R Jensen and Bjarne Toft. *Graph coloring problems*, volume 39. John Wiley & Sons, 2011.

[179] Rina Dechter. *Constraint processing*. Morgan Kaufmann, 2003.

[180] Greg Hamerly and Charles Elkan. Learning the k in k-means. In S. Thrun, L.K. Saul, and B. Schölkopf, editors, *Advances in Neural Information Processing Systems 16*, pages 281–288. MIT Press, 2004.

[181] Sanjiv K Bhatia. Adaptive k-means clustering. In *FLAIRS Conference*, pages 695–699, 2004.

[182] Daniel Aloise, Amit Deshpande, Pierre Hansen, and Preyas Popat. Np-hardness of euclidean sum-of-squares clustering. *Machine learning*, 75(2):245–248, 2009.

[183] Stuart Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982.