CDCL SAT Solver Heuristics: Clause Management, Instance Structure, and Decisions

by

Sima Jamali

M.Sc., Sharif University of Technology, 2015
B.Sc., Azad University of Mashhad, 2012

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Summer 2021

Declaration of Committee

Name:	**Sima Jamali**
Degree:	**Doctor of Philosophy**
Thesis title:	**CDCL SAT Solver Heuristics: Clause Management, Instance Structure, and Decisions**
Committee:	**Chair:** Joseph Peters Professor, Computing Science
	David Mitchell Supervisor Associate Professor, Computing Science
	Hang Ma Committee Member Assistant Professor, Computing Science
	Saba Alimadadi Examiner Assistant Professor, Computing Science
	Vijay Ganesh External Examiner Associate Professor Electrical and Computer Engineering and Computer Science University of Waterloo

Abstract

The Boolean satisfiability problem or SAT is the problem of deciding if a Boolean formula has a satisfying assignment. It was the first problem shown to be NP-complete, and remains one of the most well-known and studied NP-complete problems. We do not expect to find a polynomial time algorithm that solves all SAT problems, as this would imply equivalence of the complexity classes P and NP, which seems unlikely. However, there are algorithms and heuristics to solve SAT problems that are often effective in practice. A SAT solver is a program that takes as input a Boolean formula and tries to find a satisfying assignment for it. The most-used algorithm in SAT solvers intended for solving real-world problems is known as Conflict Driven Clause Learning, abbreviated CDCL. Due to its broad usage, improving the performance of these solvers can have a large impact on other fields that use SAT solvers and also make SAT solving a useful tool for more applications.

The practical performance of CDCL SAT solvers depends critically on a small number of key heuristic mechanisms, and work on these heuristics over the past 20 years have improved CDCL solver performance significantly. This dissertation contributes to our understanding of two of the key heuristic mechanisms, known as the decision heuristic and the clause database management scheme. There are two main foci, which are closely related. First we focus on developing light weighted methods to use measures of instance structure in solver heuristics. The structure of instances arising from real-world problems seems to be one of the main features that makes them special but there is little work exploiting structural properties within CDCL solvers. We introduce a new structural measure for SAT instances, called Centrality, and show that this measure can be used in both decision and clause management heuristics to improve solver performance. Second, we study different components of clause database management schemes in order to understand and improve them. We categorize clauses as permanent and temporary, show that the permanent set is key to solver performance and propose modifications to the criteria for permanent clauses to improve performance. In recent years, clause database management strategies used in high-performance solvers have become complex, making their study and refinement difficult. We introduce a new clause reduction scheme, called online deletion, which is simple to implement and results in comparable performance.

Keywords: Structural Properties of CNF Formulas; Simplifying Clause Deletion; Centrality in SAT; Permanent Clauses; CDCL Solvers; Heuristics in SAT Solvers

To $\mathfrak{R}$,

The light of my life!

Acknowledgements

First and foremost, I offer my sincerest gratitude to my senior supervisor, Dr. David Mitchell. I am wholeheartedly grateful for all his invaluable contributions of time and insight to make my research productive, stimulating, and exciting. Thank you, David! I would like to sincerely thank the members of my PhD dissertation committee: Dr. Hang Ma, Dr. Saba Alimadadi and Dr. Vijay Ganesh for their thorough examination of this thesis and their suggestions and remarks on this work.

A special thanks goes to my parents, a constant source of support and encouragement for me; thank you for being so loving, caring and supportive. Heartfelt thanks goes to my grandma, Sepideh, Mohamadreza and Pegah. I appreciate their endless love and support that made all these years a pleasant chapter of my life. My beloved family are always my sense of safety and calmness.

I am thankful to my friends at Simon Fraser University and my second family in Vancouver who made my studies more pleasant and enjoyable. Especially, I am grateful to Nazanin, Zahra, Mahsa, Parsian, Mona, Sina, Akbar, Mehdi, Ali, Hamid, Kiarash, Payam, Ramtin, Mina, Sajjad, Rana, Ehsan, Sepehr and Sara.

Last but not least, I am immensely grateful to my partner, my best friend and my husband, Babak! Thank you for always being on my side, for believing in me unconditionally and supporting me in this chapter of my life.

I would like to acknowledge the administrative and technical staff in the School of Computing Science for making this school a productive environment. I also wish to thank the Natural Sciences and Engineering Research Council of Canada (NSERC) for their financial support in the course of this research.

Contents

Declaration of Committee ii

Abstract iii

Dedication v

Acknowledgements vi

Table of Contents vii

List of Tables x

List of Figures xi

1 Introduction 1

1.1 SAT and CDCL Solvers . . . 1

1.1.1 CDCL Algorithm and Heuristics . . . 3

1.2 Thesis Contributions . . . 4

1.3 Thesis Structure . . . 7

2 Conflict Driven Clause Learning Solvers 8

2.1 Basic definitions and terminology . . . 8

2.1.1 Boolean Satisfiability Problem (SAT) . . . 8

2.1.2 SAT solvers . . . 9

2.1.3 Literal, Clause . . . 9

2.1.4 Conjunctive Normal Form (CNF) . . . 10

2.2 Solving Algorithms and CDCL . . . 10

2.2.1 Truth Table Method . . . 10

2.2.2 Backtracking . . . 10

2.2.3 DPLL Algorithm . . . 15

2.2.4 CDCL Algorithm . . . 17

2.3 CDCL heuristics . . . 21

2.3.1 Decision Heuristics . . . 23

2.3.2 Clause Database Management . . . 28
2.4 Summary . . . 32

3 Decision Heuristics and Instance Structure 34
3.1 Overview . . . 34
3.2 Related Work . . . 36
3.3 Structural Properties . . . 37
3.3.1 Structural Properties Computations . . . 39
3.4 VSIDS Preferences and Preferential Bumping . . . 40
3.4.1 VSIDS Preferences . . . 41
3.4.2 Preferential Bumping . . . 42
3.4.3 Preferential Bumping of Central Communities . . . 45
3.4.4 Preferential Bumping in Glucose . . . 46
3.5 Centrality based Modifications in Maple LCM Dist . . . 49
3.5.1 Decision Heuristics . . . 49
3.5.2 Performance Evaluation . . . 50
3.5.3 Performance Analysis . . . 51
3.6 Summary . . . 53

4 Clause Centrality in Deletion Strategies 54
4.1 Overview . . . 54
4.2 Clause Centrality . . . 55
4.3 Centrality based Clause Deletion in Maple LCM Dist . . . 55
4.3.1 Performance Evaluation . . . 56
4.3.2 Comparing Clauses after Deletions . . . 59
4.4 Learned Clause Quality . . . 60
4.5 Deletion Criteria in Delete-Half Schemes . . . 62
4.6 Summary . . . 63

5 Permanent Clauses 64
5.1 Overview . . . 64
5.2 PERM Set in State of the Art Solvers . . . 66
5.3 Usage in Learned Clauses . . . 67
5.4 Size and Value of PERM in MapleLCMDistChronoBT . . . 69
5.5 Varying Size and LBD Criteria for PERM . . . 72
5.6 Adding High-Centrality Clauses to PERM . . . 74
5.7 Small good clauses not to add to PERM . . . 75
5.8 SAT vs. UNSAT formulas . . . 77
5.9 Summary . . . 77

6 Simplifying Clause Database Management **79**
6.1 Overview 79
6.1.1 Performance Evaluation and Base Solver 81
6.2 Online Clause Deletion 82
6.2.1 Relating Delete-Half and Online Deletion 82
6.3 Age-Based Deletion 82
6.4 Clause Usage 83
6.4.1 Fraction Saved by RU 86
6.4.2 Clauses saved by RU and Activity 86
6.5 Clause LBD and Tier2 87
6.6 Computation Time 88
6.7 SAT vs. UNSAT formulas 89
6.8 Summary 90

7 Conclusion **91**

Bibliography **94**

List of Tables

Table 2.1 Truth Table of Formula ϕ. 11
Table 2.2 Simplifying ϕ 13
Table 2.3 Unit Propagation in solving ϕ 14
Table 2.4 Main Features of the Chosen Solvers 33

Table 3.1 Correlations amoung structure computation and solver times. 40
Table 3.2 Measures of the degree to which VSIDS prefers special variables . . . 41
Table 3.3 Effect of increasing the bump value for variables in central communities 46
Table 3.4 Comparing performance of modified solvers on a smaller benchmark . 48
Table 3.5 Performance of default Maple LCM Dist and our three modified solvers 50
Table 3.6 Number of families involved in formulas solved by our modified solvers 51
Table 3.7 Measures of search or reasoning rate for the four solvers 53

Table 4.1 Performance of Maple LCM Dist and HCdel solvers 56
Table 4.2 Measures of reasoning rate 58
Table 4.3 Measures of quality for clauses in LOCAL 59

Table 5.1 Average value of clause quality measures in solver with no clause deletion 68
Table 5.2 Learning rates in instances with Large/Small number of PERM clauses 70
Table 5.3 Performance on Satisfiable *vs* Unsatisfiable instances. 77

Table 6.1 Commonality among High-Activity Clauses and Recently-Used Clauses. 87
Table 6.2 Performance on Satisfiable *vs* Unsatisfiable Formulas. 90

List of Figures

Figure 1.1 CDCL algorithm and its main Heuristics 3

Figure 2.1 Tree showing full assignments for ϕ 12
Figure 2.2 Pruned tree using backtracking algorithm for ϕ 12
Figure 2.3 Pruned tree after setting $p = 0$ 14
Figure 2.4 Implication Graph without Conflict 18
Figure 2.5 Implication Graph with Conflict 19
Figure 2.6 Resolution on Conflict Clause 19
Figure 2.7 Further Resolution by CDCL to find an Asserting Clause 20
Figure 2.8 Evolution of Solvers 23

Figure 3.1 Computing time of structural properties 39
Figure 3.2 Mean fraction of special decisions over full run 43
Figure 3.3 Effect of preferential bumping on special decisions 44
Figure 3.4 Effect of initial *vs* uniform preferential bumping 44
Figure 3.5 Examples of community graphs with no clear coarse structure . . . 45
Figure 3.6 Examples of community graphs with coarse structure 46
Figure 3.7 Relative performance of Glucose, GLPB-Br-u and GLPB-HC-i. . . 47
Figure 3.8 Cactus plot comparing Maple LCM Dist and three modified solvers 51
Figure 3.9 Scatter plots comparing Maple LCM Dist and three modified solvers 52

Figure 4.1 Cactus plot of Maple LCM Dist and HCdel performance 57
Figure 4.2 Scatter plot of Maple LCM Dist and HCdel performance 58
Figure 4.3 Average Usage with respect to different clause quality measures . . 60
Figure 4.4 PAR-2 scores of solvers with different deletion criteria 62

Figure 5.1 Average LBD of learned clauses with respect to Age 69
Figure 5.2 PERM Size Histogram 70
Figure 5.3 Effect of removing PERM from MapleLCMDistChronoBT 71
Figure 5.4 Effect of limiting size of PERM on “LC” formulas. 71
Figure 5.5 Average size of the PERM set on LC benchmarks 72
Figure 5.6 Effect of PERM criteria on clauses and size of DB 73
Figure 5.7 Effect of PERM criterion on PAR-2 Scores 73

Figure 5.8 Performance of solvers with small clauses in Core 74
Figure 5.9 Performance of solvers with high-centrality clauses in PERM 75
Figure 5.10 Performance with PERM criteria Size$\leq$ 8 or HC. 76
Figure 5.11 Performance of solvers with new learned clauses 76

Figure 6.1 Number of TEMP Clauses in Delete-Half scheme 80
Figure 6.2 Simple Online Deletion Performance. 83
Figure 6.3 Rate of use of clauses in Local at different ages. 84
Figure 6.4 Online Deletion with Recent Usage 85
Figure 6.5 Fraction of saved clauses in different online deletion schemes . . . 86
Figure 6.6 Online Deletion With Usage and LBD 87
Figure 6.7 Fraction of Time Spent in Clause Deletion Methods 89

Chapter 1

Introduction

The Boolean satisfiability problem, or SAT, is the problem of finding an assignment to the variables of a Boolean formula that makes the formula true. A SAT solver is a software artifact that takes as input a Boolean formula (also known as a formula of propositional logic) and searches for a satisfying assignment for it. This dissertation contributes to the area of applied SAT solving, that is, the use of SAT solvers in solving problems of a combinatorial nature that come from real world applications. Hardware and Software Verification [21], Planning [62][63], Scheduling [50], Automatic test pattern generation [105] and Bioinformatics are just some examples of the problem domains where SAT solvers have been used. SAT instances coming from such applications are called "industrial".

The most effective SAT solvers for industrial instances are based on an algorithm called Conflict Driven Clause Learning, abbreviated CDCL. The performance of CDCL solvers in practice depends critically on a small number of key heuristic mechanisms. This dissertation makes several contributions to the understanding and design of two of these heuristic mechanisms, namely the decision heuristic and the clause database management scheme. The contributions are generally of three sorts. First, we study the use of instance structure in the heuristics. We introduce a new structural measure of formulas, called Centrality, and show it can be used in decision and clause database management heuristics to improve the performance of CDCL solvers. Second, we take a close look at clause database management techniques in state of the art solvers, shedding some light on the roles of their various features and ways to improve them. Third, we introduce a new simplified clause management scheme. The contributions are described in detail in Section 1.2, after we introduce the basics of CDCL solvers in Section 1.1.

1.1 SAT and CDCL Solvers

SAT is the problem of determining whether a boolean formula is satisfiable or unsatisfiable. It was the first problem shown to be NP-complete, in a 1971 paper by Stephen Cook [97, 34, 45]. A boolean formula, or just formula for simplicity, is satisfiable if there exists a

truth assignment for its variables for which the formula evaluates to true. If no such truth assignment exists, the formula is unsatisfiable. SAT has been identified as a core problem in computer science among with several other important computational problems and has been comprehensively studied, leaving a huge literature [90]. From the complexity point of view, we can not expect to find an answer for all SAT problems in polynomial time as it would imply P = NP which is believed to be unlikely. However, there are many algorithms and solvers to tackle SAT problems which are shown to be effective in practice, and as a consequence, NP-complete problems in many fields are solved by reducing or transforming them to SAT. Because of this broad usage, improving the performance of SAT solvers can have an impact on many fields and also increase the range of applications for which it is useful. In this dissertation we will focus on practical SAT research and solving SAT problems in CNF format with solvers that have CDCL as their core algorithm, that is, CDCL SAT solvers.

The CDCL algorithm was proposed by Marques-Silva and Sakallah in 1996 as an extension to the DPLL algorithm for SAT [98]. At the time, DPLL was not really useful in practical settings but was theoretically interesting [9]. The CDCL algorithm, introduced in the solver GRASP [98, 78] changed the picture a lot. GRASP empirically showed that solving real world problems using a SAT solver can be practical. Since that time, performance of SAT solvers has been improved by orders of magnitude in many practical settings using methods like activity-based heuristics [39] and LBD-based clause deletions [9]. Due to its remarkable efficiency, specifically in practical settings, CDCL has been the core algorithm in most SAT solvers today to the extent that the term "CDCL solver" is often used synonymously with "SAT solver". Modern CDCL solvers are able to solve real world problems with millions of variables in reasonable time, which is impressive considering their worst case running time is exponential in the number of variables. Since 2002 there have been yearly SAT Solver competitions to evaluate SAT solvers and they continuously show performance improvements each year [1]. It seems like currently, only CDCL solvers are able to win in the application or industrial competition track [59]. The focus of this work will be on performance of CDCL solvers on benchmark formulas from the application track of the competitions.

The widespread use of SAT solvers in many fields in both academia and industry has led the community to provide reliable, robust, open source and easily embeddable solvers. Some of the most influential CDCL solvers over the past 20 years are GRASP [98], Chaff [81], MiniSAT [39], Glucose [8], Lingeling [20], MapleSAT [71], and Cadical [22] to date. One of the important ways these solvers differ is in the design and implementation of their key heuristics. Over the years, these heuristics have been built upon each other and improved the performance of solvers significantly.

1.1.1 CDCL Algorithm and Heuristics

The main solving scheme in all CDCL solvers is the same. They get a CNF formula as input and start a search to find a satisfying assignment for it or prove its unsatisfiability. This starts by picking a variable in the formula, assigning a truth value to it and performing Boolean Constraint Propagation (BCP). This will be repeated until the solver reaches a conflict (the partial assignment constructed so far makes a clause false) or satisfies all clauses. If all clauses are satisfied, the solver will report the formula is satisfiable. If a conflict is reached, the solver will learn a clause and add it to the clause database to prevent possible similar conflicts in future. Then it backtracks to a level in the search tree before this conflict happened and continues by picking another unassigned variable and repeats the process. If a backtrack is to level 0, the solver will report the formula is unsatisfiable and stop the search.

The main CDCL algorithm has been the same ever since it was first introduced in 1996 and can be found in most state of the art solvers but with changes to a number of details and heuristics. Three main CDCL heuristics are known as the Decision Heuristic, the Clause Database Management Scheme and the Restart Policy. Figure 1.1 illustrates the CDCL algorithm and where its most important heuristics play their roles. A restart is a backtrack to level zero where all variables become unassigned and the solver starts a new and different search. The restart policy determines when this happens. Our focus in on the other two heuristics.

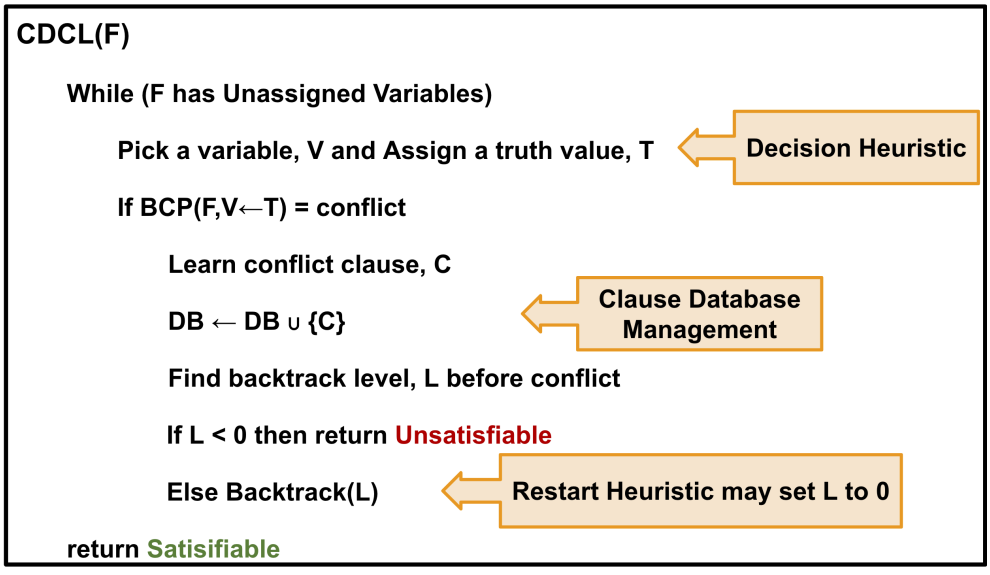


Figure 1.1: CDCL algorithm and its main Heuristics

The decision heuristic determines which variable will be chosen next to assign a value to, and which truth value will be assigned. This has a considerable impact on the performance as it determines which branches of the search tree to explore first. One of the most popular

decision heuristics that solvers use is called "Variable State Independent Decaying Sum" or VSIDS. Some version of it can be found in all state of the art solvers today [81]. VSIDS tries to keep the search in parts of the formula that solver is focusing on so when choosing the next decision variable, it gives more priority to the ones that were involved in recent conflicts. Recently a new decision heuristic called "Learning Rate Based" or LRB was introduced and can be found in some recent solvers [69, 53]. LRB aims to prioritize variables used in recent conflicts as well but has a different approach based on machine learning techniques to do so [69]. Both VSIDS and LRB use an activity-based scoring system for variables and pick the unassigned variable with highest score at each time.

As suggested by its name, Clause Database Management is responsible for managing the number of clauses in the database and how they are stored. Modern CDCL solvers can learn and add thousands of clauses to the database per second which can ruin the solver's performance very fast [87]. The reason is that not only keeping clauses consumes memory that can eventually result in exhaustion of the available memory, but also it is costly to perform BCP on clauses and not all clauses are valuable enough to be worth the cost [89]. Clause Database Management determines which clauses should be removed from the database, when this should happen and how. In particular, most learned clauses must be deleted to keep the clause database of practical size which is why this heuristic is also known as the clause deletion heuristic [9]. There are two main aspects to a clause deletion scheme. The first is a method to categorize clauses as likely to be useful in future (high quality), or not (low quality). The aim is to keep the high quality clauses in the database and delete from the low quality ones. The second is implementation of an algorithmic method to remove low quality clauses efficiently. There are some known cause quality measures that are believed to predict clauses usefulness to some extent and usually a combination of some or all of them is utilized in a solver's clause deletion scheme. Size, LBD, Usage (Activity) and Age are probably the most important. The most popular scheme for deleting low quality clauses is to periodically sort the clauses in the learned-clause database based on a quality measure and delete some (usually half) of the lowest quality. This scheme, which we call Delete-Half, has been very widely used in CDCL solvers for many years [9, 104, 71]. Typical clause database management strategies involve two stores of learned clauses in the database. One is for keeping the clauses that are believed to be too important to delete and they are kept in the database permanently. The binary clauses usually belong to this store. The rest of the clauses are considered temporary and clause deletion happens among them [58].

1.2 Thesis Contributions

This research introduces a novel measure in the context of SAT solving, Centrality which is based on the structure of SAT formulas and proposes different decision and clause database management heuristics to utilize this measure. We illustrate the effectiveness of using Cen-

trality by adopting these heuristics in state of the art solvers and show performance improvements. We also report many experimental studies to help illuminate the behaviour of heuristics, primarily in clause database management. We review the current clause database management schemes to understand different components of them better, improve them further and propose a simpler method to replace the current deletion scheme. In summary, our contributions are as follows.

- **Centrality in SAT formulas:** Structure of the formulas in industrial SAT instances seems to be one of the main features that makes them special and can help explain the fact that CDCL solvers perform very well on them in comparison with much smaller instances that are designed to be hard. There are limited works on exploiting structural properties of formulas in the solver and in many of them, extracting the properties is very expensive so it is hard to use them in solvers without damaging their performance. We introduce betweenness centrality, a measure which is extracted from the primal graph of CNF formulas. The betweenness centrality of a vertex v in a graph G is the number of shortest paths between pairs of distinct vertices that visit v. The betweenness centrality of a variable v in formula ϕ is the betweenness centrality of v in the un-weighted primal graph $G(\phi)$. We show that by using the Brandes algorithm [28], computing centralities can be done efficiently for a large number of industrial formulas and by using sampling, we can further calculate good approximate centrality values even more efficiently. We examine the behaviour of the VSIDS decision heuristic experimentally and show that this heuristic favors variables that have high centrality values (and some other structural properties) which may help explain their efficiency on industrial formulas.

- **Preferential Bumping in decision heuristics:** Initially, we show that VSIDS decision heuristic in the state of the art solvers favors some special groups of variables categorized by their structural properties without actually aiming for those properties in the first place. We introduce a new scheme to be added to the decision heuristics in the solvers that increase the focus of the decisions on a special set of variables which we call Preferential Bumping. Using preferential bumping, we modify different CDCL solvers to choose more decisions among the variables of a special set and we show that preferential bumping of high centrality variables can improve the performance of solvers. We use a similar method to change the preference in the LRB decision heuristic. In particular, preferential bumping of high centrality variables clearly improved the performance of Maple LCM Dist, which took first place in the industrial category of the 2017 SAT solver competition.

- **Exploiting centrality as a clause quality measure:** Considering the positive results we have from utilizing centrality in decision heuristics, we show how it can also be used in the clause database management heuristic. We introduce a new clause

quality measure called Centrality which is defined as the average betweenness centrality of variables in a clause. We show that by implementing centrality-based deletion schemes in state of the art solvers, we can improve their performance. In particular, the performance of Maple LCM Dist, which was the first place solver in the industrial category of the 2017 SAT solver competition, was improved significantly. We also compare centrality with three other main clause quality measures, LBD, Size and Activity in deletion strategies and make various observations. We show that Centrality of remaining clauses after deletion is the only quality measure that perfectly correlates with the speed of the solvers. We show that clauses with higher centrality values are more useful in conflict analysis which is similar to the behaviour of small or low-LBD clauses. We also compare using various clause quality measures in deletion and show that in the presence of a permanent set of clauses with low LBD, centrality based deletion results in the best performance. These experiments apply to the MapleSAT solvers with a 3-tier clause database management heuristic which is a common scheme in recent state of the art solvers.

- **Categorizing clauses to PERM and TEMP in clause management strategies and illustrating the importance of PERM:** The clauses learned by the solver can be partitioned into two categories with respect to deletion. The set of clauses that are never considered for deletion by the clause database management heuristic and are stored permanently we call PERM. The set of clauses that might get deleted we call TEMP. Historically PERM usually is the set of binary clauses, which are believed to be very valuable. In recent years, other clauses with small LBD also sometimes belong to PERM. We study the importance of PERM in recent solvers and show that even though by relaxing the criteria for PERM clauses the solver ends up saving millions of clauses in some cases, they are still valuable and our experiments suggest they should not be removed. We also show that clauses with high centrality, which tend to be more useful in the conflict analysis, make good candidates for PERM. We modify a recent version of MapleSAT and its clause deletion scheme to add high centrality clauses to PERM and show that this improves performance. In particular, modifying MapleLCMDistChronoBT, the winner of the SAT competition 2018 on the industrial formulas to keep small clauses (of size at most 8) permanently or to add high centrality clauses to its PERM set, improved its performance significantly. Finally, we introduce a new scheme to learn additional small clauses with low cost. We show they can be beneficial to solver's performance if added to TEMP but not PERM.

- **Proposing a simple and novel clause deletion scheme called Online Deletion:** The clause database management heuristic is usually considered in two aspects. One is having good clause quality measures to keep the more useful clauses and the other is efficient clause deletion algorithms to remove the rest. Over time many refine-

ments have combined to make the overall mechanism in the best recent solvers quite complex. This complexity makes it hard to evaluate the contributions of individual elements, and is an obstacle to adding new features or refined quality measures. We take a closer look at the recent heuristics for clause database management and their complexity to discuss ways we can possibly simplify this heuristic. We also discuss other unappealing features of current deletion algorithms which involve periodically sorting and deleting half the clauses with lower qualities. Examples are constantly changing the size of clause database or giving different clauses different chances to be kept without considering their quality. We propose a new deletion scheme, which we call Online Deletion and illustrate how this simple scheme can overcome those problems while keeping the performance of the solver at the same level. To aid in understanding the degree to which the particular methods play a role in solver performance, we present data from a number of experiments measuring performance or other properties.

1.3 Thesis Structure

The remainder of this thesis is organized as follows. Chapter 2 explains in detail the basics of the CDCL algorithm and its most important heuristics. In chapter 3 we introduce some structural properties of SAT formulas, and introduce betweenness centrality of variables. We also introduce methods for exploiting these properties by modifying standard CDCL decision heuristics. In chapter 4, we introduce clause centrality, provide evidence of its usefulness as a clause quality measure, and show that solver performance can be improved by utilizing clause centrality in clause deletion schemes. In chapter 5, we study the roles of permanent (PERM) and temporary (TEMP) clauses in solvers. We study the use of different kinds of clauses in PERM and the effects on performance, and in particular show that adding high centrality clauses to PERM can improve solver performance. In chapter 6, we explain a new and simple clause deletion algorithm, called Online Deletion and discuss its advantages. Chapter 7 concludes.

Chapter 2

Conflict Driven Clause Learning Solvers

2.1 Basic definitions and terminology

In this section we start by reviewing terminology used in the literature and proceed to discussing the main algorithms and techniques for solving SAT problems.

2.1.1 Boolean Satisfiability Problem (SAT)

In Boolean formula is a formula of propositional logic, composed of Boolean variables are connectives for the standard Boolean logic operators, negation ($\neg$), conjunction ($\wedge$) and disjunction ($\vee$). A Boolean variable can be assigned to either TRUE or FALSE. A truth assignment for a formula is an assignment of a truth-value (either true or false) to variables of the boolean formula. If the truth value of all the variables is set, we call it a full assignment. It is called a partial assignment when the truth value of some variables remains unassigned [40].

A boolean formulas is considered Satisfiable if there exist a truth assignment for its variables in a way that it evaluates to true. If no such truth assignment exists, the formula is Unsatisfiable. A Boolean satisfiablity problem is the problem of determining whether the formula is Satisfiable or Unsatisfiable. SAT solvers are widely used to find the answer for this problem. We say that two formulas are logically equivalent if they have the exact same satisfiabing assignments. It means that any truth assignment that makes one formula true (false), will make the other one true (false) as well. A weaker definition of this form is equisatisfiability. Two formulas are considered equisatisfiable if they are both satisfiable or they are both unsatisfiable. Any two formulas that are logically equivalent are also equisatisfiable but the converse does not hold. It means that there may exist a specific truth assignment that satisfies one formula but not the other [65].

2.1.2 SAT solvers

Boolean satisfiability problem or SAT is a well-known NP-complete problem [97][34][45]. Given a possible satisfying assignment, its correctness can be determined in polynomial time by evaluating the formula but there is no known polynomial time algorithm to find a satisfying assignment for a formula. In fact given that any NP-complete problem can be relatively easily transformed into another one, if there was such a solution then all NP-complete problems could be solved in polynomial time which would result in P=NP. All known solutions for NP-Complete problems that are currently being used, in the worst case, require runtime that grows exponentially with the size of the instance. For example, the worst case for finding a satisfying assignment (or determining there is none) for a formula with n variables is to check all the 2^n possible assignments. However in the past years, many algorithms and heuristics have been introduced for solving SAT problems that usually require time much less than the worst case. Because of the efficiency of algorithms used in SAT solvers, they have become known as a go-to tool for solving NP-complete problems in general. Hardware and Software Verification [21], Planning [62][63], Scheduling [50], Automatic test pattern generation [105] and Bioinformatics are just some examples of the problem domains that use SAT solvers in their problem solving. In this chapter, we go over some definitions and address some basic algorithms for solving SAT problems.

2.1.3 Literal, Clause

Let us denote a variable by x. A literal l is a variable or its negation (specified with $\neg$). We say that a literal l has positive polarity if x appears positively in l ($l = x$) and that it has negative polarity if x appears negatively in l ($l = \neg x$). Literals l_1 and l_2 are complements when one is a negation of the other. A clause is a logical sentence consisting of a finite number of literals connected with disjunctions. A clause is satisfied if at least one of its literals is true and it is falsified if all its literals are false. A clause, C, of size n can be written in the following format where l_i are its literals: $C = (l_1 \vee l_2 \vee ... \vee l_n)$. Alternatively, we can use a set notation to show clause C as follows where we clause C is shown as a set that contains n literals: $C = l_1, l_2, ..., l_n$.

The size of a clause, |C|, is the number of literals in it. An empty clause is a clause of size 0 and is denoted by several symbols like $\emptyset, \perp, or \square$. The truth value of an empty clause is always false and so we say its unsatisfiable. In the context of SAT, the presence or derivation of the empty clause is a proof of unsatisfiability. A clause of size one, which contains only one literal, is called a unit clause. In order to satisfy a unit clause, its only literal, l, has to be set to true. Therefore a satisfied unit clause represents a fact meaning its variable, x, has to be true (false) if $l = x (l = \neg x)$.

2.1.4 Conjunctive Normal Form (CNF)

A formula in a boolean satisfiablity problem may be expressed in particular structures in special cases. Conjunctive Normal Form or CNF is one of these structures and almost all SAT solvers expect the input boolean formula to be in CNF format. A CNF formula is a conjunction of clauses. It can alternatively be shown as a set of clauses in the set notation. There are different methods in boolean algebra that can transform any propositional boolean formula to an equisatisfiable formula in conjunctive normal form. The Tseitin transformation is one these methods that can do it in polynomial time with a linear increase in size of the formula. [107]

2.2 Solving Algorithms and CDCL

We start by looking at some basic algorithms for solving SAT problems in CNF format. Note that not all of them need to get their input in this format but from now on, when we say formula, we assume its in CNF format.

2.2.1 Truth Table Method

We start by the truth table method in which to determine if a formula is satisfiable, we build the truth table for the formula to find satisfing assignments or show there are none. To build the truth table, we add a row for each possible truth assignment of the variables in a formula. Given that each variable can be assigned to either true or false, for a formula with n variables in it, there will be 2^n rows. Even though this approach is sound and complete as it checks all possible truth assignments, the size of the table grows exponentially and it is not practical for large formulas.

Let's assume we have the following formula: $\phi = \{(p, q)(\neg q, r)(p, \neg r)\}$. ϕ has 3 variables which means there will be $2^3 = 8$ rows in the truth table. Now for each truth assignment (row), the truth value of each clause will be determined separately using reduction rules. ϕ is satisfied only when all its clauses are satisfied. After determining the truth value of each clause, we can find the truth value of ϕ for each row. ϕ is considered a satisfiable formula iff its value is true in at least one row of the truth table. It is unsatisfiable otherwise. Table 2.1 shows the truth table for formula ϕ. Note that we use 1 to show that a variable or formula is assigned to true and 0 to show that it is assigned to false. As there are 3 truth assignments that sets ϕ to 1(true), then ϕ is a satisfiable formula.

2.2.2 Backtracking

As you can see, the truth table method examines all possible full assignments of the variables in the formula. To determine the satisfiability of a formula, it is not always necessary to

p	q	r	(p, q)	$(\neg q, r)$	$(p, \neg r)$	ϕ
0	0	0	0	1	1	0
0	0	1	0	1	0	0
0	1	0	1	0	1	0
0	1	1	1	1	0	0
1	0	0	1	1	1	1
1	0	1	1	1	1	1
1	1	0	1	0	1	0
1	1	1	1	1	1	1

Table 2.1: Truth Table of Formula ϕ.

do that. For example in the previous formula, we can show ϕ is satisfiable by a partial assignment of setting p and r to true even without knowing the value of q. In this case, the amount of time we could be saving by not examining q is small but in larger formulas with many variables, it can save a lot of time and space. Similarly, the unsatisfiability of some formulas can be detected by partial assignments.

In a simple backtracking approach we aim to search the truth assignment space of a formula by assigning truth values to variables one at a time. To represent the search space in this approach we use a binary tree. The root of the tree represents the empty assignment when none of the variables are assigned to a truth value. At each node a variable can be assigned to two different values so there are two branches. The depth of the tree can be as big as the number of variables in the formula. The middle layers of the tree are showing partial assignments whereas each leaf is showing a full assignment. The backtracking algorithm at each step, assigns a value to a variable and checks if the formula is satisfied. At that point if the formula is satisfied, it can complete the partial assignment by arbitrary assigning the remaining variables, reporting one or more satisfying assignments for the formula and terminating the algorithm. Alternatively if the formula evaluates to false at a point, the algorithm will not continue assigning more values in that branch and blocks that path as all of them will reach to an assignment/leaf that is not satisfying. Instead, it backtracks to the most recent decision and continues on a different branch. The algorithm continues its work until either a satisfying is found or all possible assignments (unblocked paths) have been explored and it will report the unsatisfiability of formula.

Going back to our example $\phi = \{(p, q)(\neg q, r)(p, \neg r)\}$, figure 2.1 shows a full assignment of its 3 variables with the binary tree representation. As mentioned each leaf shows a full assignment and here we have 8 leaves for ϕ. Figure 2.2 on the next page illustrates how backtracking algorithm can prune the search space for the same formula. Consider the first two assignments that sets variables p and q to false (0). Just by setting these two variables and evaluating formula ϕ, the algorithm will determine that this formula is unsatisfiable under this partial assignment because one of its clauses, $(p \vee q)$, is already made false. At

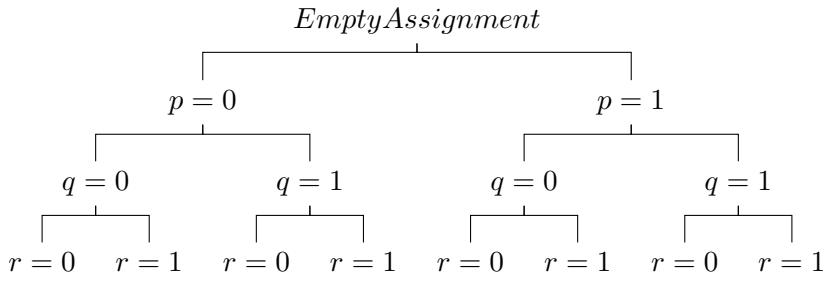


Figure 2.1: Tree showing full assignments for ϕ

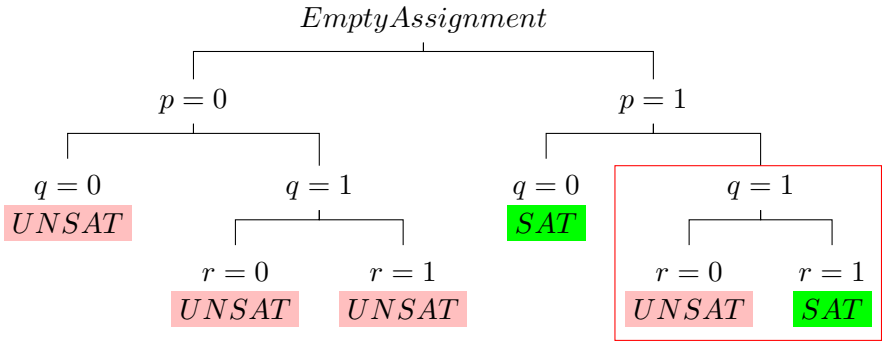


Figure 2.2: Pruned tree using backtracking algorithm for ϕ

this point the algorithm blocks the current path and marks it as unsatisfiable (UNSAT) and backtracks to the most recent decision ($p = 0$) and tries a different branch ($q = 1$). We say that the search space has been pruned as the algorithm will not need to explore the blocked subtree after this. A similar situation happens in the path where p is set to true and q is set to false. Evaluating the formula at this point results in showing its satisfiability under this partial assignment because all clauses in ϕ are satisfied. Hence the algorithm would know any arbitrary assignment to r will still make the formula satisfied and marks this node as satisfiable (SAT). The algorithm terminates at this point reporting the formula as satisfiable as it already found two satisfying assignments for the formula ϕ ($p = 1, q = 0, r = 0$ or $p = 1, q = 0, r = 1$). Alternatively, we can let the algorithm continue examining the rest of the tree to find all satisfying and non-satisfying assignments. The sub-tree illustrated with a rectangular box in figure 2.2 is found by the algorithm after the first satisfying assignment was found and will only be explored if the algorithm is designed to find all satisfying assignments. The pruned space in this example is not that significant but you can imagine if the search space is large and this happens in top layers of the tree, this algorithm can potentially save a lot of space and time.

Original formula	Set $p = 0$
(p, q)	(q)
$(\neg q, r)$	$(\neg q, r)$
$(p, \neg r)$	$(\neg r)$

Table 2.2: Simplifying ϕ after setting $p = 0$

Unit Propagation and BCP

Now, we explain two techniques that can be added to the simple backtracking algorithm for further pruning of the search space. Consider the backtracking algorithm and binary tree representation. At each branch after assigning a truth value to a variable, we can simplify the formula. If a variable x is assigned to true, all clauses with literal $l_1 = x$ in them are satisfied and can be eliminated from the formula for further consideration on that branch. Also, all clauses containing the literal $l_2 = \neg x$ can be modified by removing l_2 from them because in order for those clauses to become satisfied, one of their remaining literals must become satisfied. In other words if x is true, any clause C_1 containing $\neg x$ is equisatisfiable to a clause $C_2 = C_1/\neg x$. Similarly, if variable x is assigned to false, all clauses with literal $l_2 = \neg x$ can be removed from the formula and all clauses containing the literal $l_1 = x$ can be modified by removing l_1. To illustrate this method, lets go back to our example formula $\phi = \{(p, q)(\neg q, r)(p, \neg r)\}$. Table 2.2 shows the simplified formula after setting $p = 0$ (first branch in the tree). Let's call this new simplified formula ϕ', We can say that ϕ' is equisatisfiable to ϕ after setting $p = 0$.

Simplification enables us to use a very important technique called Unit Propagation that can prune the search space significantly. Whenever there is a clause of size 1 in the formula, we know that all the satisfying assignments of the formula must set the literal in that clause to true because that's the only way we can satisfy the clause. These clauses of size one are called unit clauses and represent a fact that its literal has to be true for the formula to be satisfied. Considering the tree representation, at any point that there is a unit clause in the simplified (or original) formula, the sub-tree that falsifies that clause can be pruned without any further computation. Simplifying the formula by removing literals and using unit clauses is called unit propagation or the unit clause rule.

Let's go back to our example. As shown in table 2.2, after setting $p = 0$ and simplifying the formula, two of the clauses become unit ((q) and $(\neg r)$). Using unit propagation, we know that in order to satisfy the formula, both literals in them should become true so $q = 1$ and $r = 0$ and we can prune any branch in the tree that has different values for them. Figure 2.3 shows the pruned tree where sub-trees in red rectangles are removed from the search.

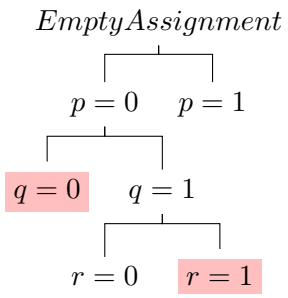


Figure 2.3: Pruned tree after setting $p = 0$

Original formula	Set $p = 0$	Unit Propagation sets $q = 1$	Unit Propagation sets $r = 0$
(p, q)	(q)	-	-
$(\neg q, r)$	$(\neg q, r)$	(r)	()
$(p, \neg r)$	$(\neg r)$	$(\neg r)$	-

Table 2.3: Unit Propagation in solving ϕ

Table 2.3 illustrates how the formula ϕ is simplified after setting $p = 0$ and applying unit propagation. Note that applying unit propagation can make more unit clauses and trigger further unit propagation.

Boolean Constraint Propagation (BCP) refers to the process of applying unit propagation to a boolean formula until no further unit propagation is possible and a fix point is reached [112]. This means until there is no more unit clauses in the formula or an empty clause has been created or derived. Deriving an empty clause means that the formula is unsatisfiable under the given truth assignment. For example table 2.3 shows that after assigning the variable p to false which is a partial assignment for variable of ϕ and applying unit propagation, an empty clause is derived that appears in the second column of the table. This means the formula ϕ is unsatisfiable if p is set to false. Using the backtracking algorithm, we can backtrack to the node where p was assigned to false (root of the tree in this case) and try a different branch by setting p to true.

Resolution

Resolution [93] in propositional logic is a rule of inference that produces a new clause implied by two clauses containing literals that are complements of each other. Let C_1 be a clause $\{l, a_1, a_2, ..., a_n\}$ and C_2 be another clause $\{\neg l, b_1, b_2, ..., b_m\}$. Using resolution, a new clause C_3 can be derived as $C_3 = (C_1 \cup C_2) \backslash \{l, \neg l\}$ which is the disjunction of all literals in C_1 and C_2 excluding l and $\neg l$. The resolution rule states that if C_1 and C_2 are satisfied under some satisfying assignment then C_3 must also be satisfied. Resolution can be depicted in the following format.

$$\frac{C_1{:}\{l,a_1,a_2,...,a_n\} \qquad C_2{:}\{\neg l,b_1,b_2,...,b_m\}}{C_3{:}\{a_1,a_2...,a_n,b_1,b_2,...,b_m\}}$$

We say that clause C_3 is the resolvent of clauses C_1 and C_2. Semantically, this rule is deriving a logical conclusion of considering both cases of setting l to true and false. In the case that l is true, consequently clause C_1 is satisfied. Since $\neg l$ is false, in order for C_2 to be true at least one of its literals $\{b_1, b_2, ..., b_m\}$ should be true which results in clause C_3 being satisfied. Similarly in the case that l is false ($\neg l$ is true), clause C_2 is satisfied and for clause C_1 to be satisfied, at least one of it literals $\{a_1, a_2, ..., a_n\}$ must be true. This will result in C_3 being satisfied. This rule combined with a complete search algorithm for SAT problems results in a sound and complete approach in [93] and makes resolution a very powerful tool in boolean satisfiability. Note that unit propagation is a special case of the resolution rule where one of the clauses is unit.

2.2.3 DPLL Algorithm

The Davis-Putnam-Logemann-Loveland (DPLL) algorithm [36] (which is an variation of the Davis-Putnam algorithm [37]) was introduced in 1962 by Martin Davis, George Logemann and Donald Loveland (DPLL is also known as DLL) and is used to determine the satisfiability of a CNF formula. It has a more general application as well but we remain focused on propositional CNF formulas in this report. DPLL is originally a recursive algorithm and iteratively chooses one variable and splits the formula into two smaller formulas with a unit clause, one containing the variable in negative form and the other with the variable in positive form. It is called *Split Rule* and is similar to setting the variable to false and true at each branch as shown in the backtracking algorithm before. After assigning a value to the variable, the formula is simplified by BCP and the process is repeated for these smaller formulas. If an empty clause is derived at some point, the algorithm terminates and reports unsatisfiability. If DPLL assigns all variables without deriving an empty clause, the formula is proven to be satisfiable. This gives us an efficient approach to solve SAT problems that even today after almost 60 years, is the core algorithm used in modern SAT solvers.

Algorithm 1 Davis Putnam Logemann Loveland Algorithm

```
1: procedure DPLL(φ)
2:     if ∅ ∈ φ then                                  ▷ Empty Clause Derived
3:         return Unsatisfiable
4:     if φ = ∅ then                                  ▷ All variables eliminated
5:         return Satisfiable
6:     BCP(φ)                                         ▷ Boolean Constraint Propagation
7:     Pick a variable x ∈ φ
8:     return (DPLL(φ ∪ {x}) ∨ DPLL(φ ∪ {¬x}))        ▷ Split Rule
```

Algorithm 1 illustrates the steps in DPLL. This algorithm is a recursive version of DPLL however in practice an iterative version is used in SAT solvers. Also, since the generated subformulas are relatively large they are not saved by the algorithm. Instead of executing elimination of clauses and literals by changing the input formula, the solvers keep a stack of partial assignments to variables at all levels and as a result, the current state of the formula can always be calculated as needed.

Algorithm 2 Davis Putnam Logemann Loveland Algorithm - Iterative Version

```
1:  procedure DPLL(φ)
2:      level ← 0
3:      loop
4:          BCP(φ)
5:          if Conflict then          ▷ Formula is Unsatisfiable under current assignment
6:              if level == 0 then
7:                  return Unsatisfiable
8:              level ← level − 1
9:              Backtrack(φ, level)
10:             Decision[level].value() ← true          ▷ Change the last decision to true
11:         else
12:             if φ = ∅ then
13:                 return Satisfiable
14:             Pick an unassigned variable x ∈ φ          ▷ Next variable to branch on
15:             level ← level + 1
16:             Decision[level].var() ← x
17:             Decision[level].value() ← false          ▷ Try setting x to false first
```

Algorithm 2 shows an iterative version of DPLL that is closer to what is used in practical SAT solvers. It is very similar to the backtracking algorithm mentioned before along with the unit propagation rule. It starts from Level 0 which is the root in the tree representation. At each iteration, it applies BCP until a fixed point is reached. This means either an empty clause is reached ($Conflict$) or no more unit clauses exist in the formula. If a $Conflict$ is reached, it means the formula is unsatisfiable under the current assignment. So the algorithm changes the last decision and tries a different branch or reports unsatisfiability in case it is at level 0 (no more decisions can be made). If BCP doesn't report a $Conflict$, the algorithm picks the next variable to branch on and moves to the next level. In this report, we call the branching variable the *Decision Variable* and call the levels in the algorithm, *Decision Levels*. The process of picking decision variables will continue until all variables are assigned and the algorithm reports Satisfiability. Remember that the iterative version of DPLL does not remove the literals from the clauses to make them unit. A clause is considered unit if it is unit under the current assignment. For example consider a clause $C_1 = \{\neg x, y\}$ where x is assigned to $true$. C_1 is considered unit under this assignment because we can actually derive clause $C_2 = \{y\}$ by applying unit propagation. The *Backtrack* function in DPLL

not only removes the assignments made to last decision variable, it also undoes the BCP process and all the assignments made by BCP in the last level.

2.2.4 CDCL Algorithm

Almost 30 years after DPLL was first introduced, a powerful extension to it called Conflict Driven Clause Learning (CDCL) was implemented in the solver, GRASP [99, 78]. This algorithm was shown to be very efficient for solving large real world SAT formulas and was adopted quickly by solvers to the extent that today, almost all modern SAT solvers working on industrial formulas use the CDCL algorithm. These solvers are also referred to as CDCL solvers. The power of CDCL comes from its ability to learn new clauses after conflicts that are added to the formula. These clauses are logical consequences of the original formula which means they are not mandatory and can be deleted at any time. Another important difference of CDCL with simple DPLL is its non-chronological backtracking which also helps with pruning the search space.

Intuitively speaking, after each conflict, CDCL learns at least one reason for that conflict in the form of a new clause. These clauses are also called *learned clauses*. For example, assume a conflict happens after assigning variables x_1 to x_{10} all to $true$ at decision levels 1 through 10. After analyzing this conflict, CDCL learns a new clause $C_1 = \{\neg x_1, \neg x_3, \neg x_{10}\}$ as a reason for it. This means that at least one of these three variables should be assigned to $false$ in order to satisfy the formula. Now, CDCL introduces non-chronological backtracking. Assuming this clause was in the formula earlier, after decision level 3, C_1 would have been a unit clause because x_1 and x_3 are both $true$ at that level. This means that by unit propagation x_{10} would have been assigned to $false$ to satisfy C_1. Therefore, after learning C_1, CDCL directly backtracks to level 3 and since C_1 is now in the formula and unit under current assignment, the algorithm immediately assigns x_{10} to false by unit propagation. Since this clause is asserting the value of x_{10} at level 3, it is also referred to as the *Asserting Clause*. In principle CDCL can learn many clauses after each conflict but modern solvers learn only one clause [6]. In earlier solvers like GRASP [78], CDCL learns more than one clause per conflict. Algorithm 3 shows the DPLL algorithm extended with CDCL. For simplicity, we call this algorithm CDCL from now on.

The main difference between algorithms 3 and 2 is in the *AnalyzeConflict()* function. In CDCL, after each conflict a new clause C is learned and a possibly non-chronological backtracking level, $level$, is reported by this function. To explain this process in CDCL better, we first show the variable assignments with a directed acyclic graph (DAG) referred to as the *Implication Graph*.

Algorithm 3 DPLL Extended with CDCL

```
1: procedure CDCL(φ)
2:     level ← 0
3:     loop
4:         BCP(φ)
5:         if Conflict then                ▷ Formula is Unsatisfiable under current assignment
6:             if level == 0 then
7:                 return Unsatisfiable
8:             (level, C) ← AnalyzeConflict()                    ▷ Learns clause C
9:             φ ← (φ ∪ C)
10:            Backtrack(φ, level)
11:            Decision[level].value() ← true          ▷ Change the last decision to true
12:        else
13:            if All variables assigned then
14:                return Satisfiable
15:            Pick an unassigned variable x ∈ φ              ▷ Next variable to branch on
16:            level ← level + 1
17:            Decision[level].var() ← x
18:            Decision[level].value() ← false                ▷ Try setting x to false first
```

Implication Graph and Analyzing Conflicts

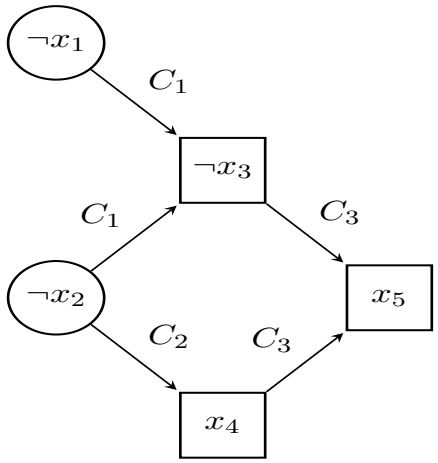


Figure 2.4: Implication Graph without Conflict

Figure 2.4 shows the implication graph for the formula $\phi_1 = \{C_1, C_2, C_3\}$ where $C_1 = \{x_1, x_2 \neg x_3\}$, $C_2 = \{x_2, x_4\}$ and $C_3 = \{x_3, \neg x_4, x_5\}$. Each node in the implication graph represents a variable assignment by the algorithm. The elipse shaped nodes show decision variables and the rectangular nodes show variable assignments done by BCP. The variables are in both negative and positive forms which shows the polarity of the truth assignment for those variables. In this example, CDCL picks x_1 and x_2 respectively at level 1 and 2 as decision variables and sets them to $false$. Next, (still in level 2) the BCP is activated and

using C_1 which is a unit clause under this assignment, x_3 is also set to $false$. It is shown in the implication graph by adding an edge from both x_1 and x_2 to x_3 with the label C_1. This label indicates that clause C_1 is the *reason* for assigning x_3 to false. Similarly x_4 and x_5 are both set to *true* using clauses C_2 and C_3 as reasons. Since all variables are assigned without any conflict, the formula is found to be satisfiable at level 2.

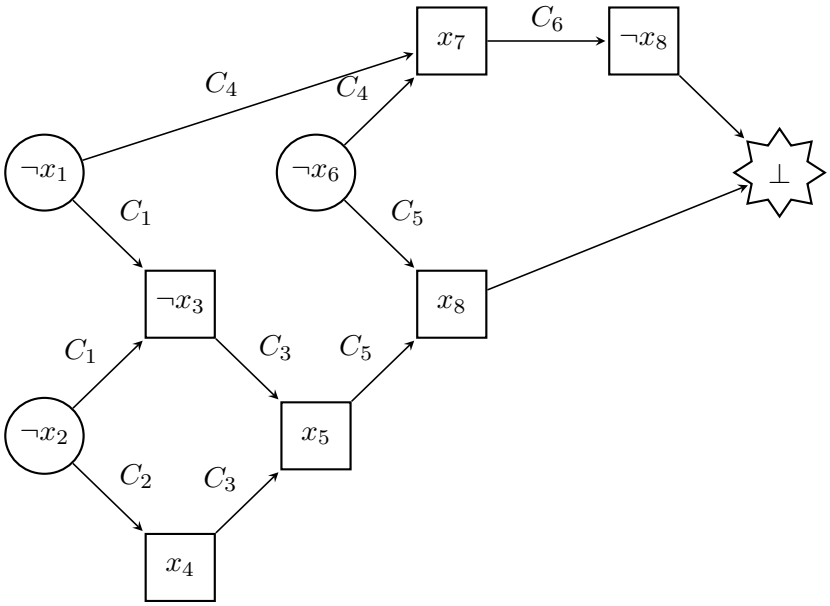


Figure 2.5: Implication Graph with Conflict

Now lets look at an example with conflict. Consider formula $\phi_2 = \{C_1, C_2, C_3, C_4, C_5, C_6\}$ where $C_1 = \{x_1, x_2 \neg x_3\}$, $C_2 = \{x_2, x_4\}$, $C_3 = \{x_3, \neg x_4, x_5\}$, $C_4 = \{x_1, x_6, x_7\}$, $C_5 = \{\neg x_5, x_6, x_8\}$ and $C_6 = \{\neg x_7, \neg x_8\}$. As before, CDCL picks x_1 and x_2 respectively at level 1 and 2 as decision variables and sets them to $false$. Next variables x_3, x_4 and x_5 will be assigned by BCP using C_1, C_2 and C_3 as reasons. Going to level 3, the algorithm picks x_6 as decision variable and sets it to $false$. Using clauses C_4 and C_5 both variables x_7 and x_8 will be assigned to *true* at this point. Having x_7 set to *true*, in order to satisfy C_6, x_8 needs to be set to $false$ which creates a conflict because it has been assigned to *true* previously and C_6 is falsified. We call this clause a conflicting clause because it is the reason for this conflict. This conflict is shown by $\perp$ in the implication graph in figure 2.5.

$$\frac{C_5{=}\{\neg x_5{,}x_6{,}x_8\} \qquad C_6{=}\{\neg x_7{,}\neg x_8\}}{C_7{=}\{\neg x_5{,}x_6{,}\neg x_7\}}$$

Figure 2.6: Resolution on Conflict Clause

At this point, we want to know why a conflict happened under the current truth assignment so CDCL calls the *AnalyzeConflict()* process. We briefly explain how this process works. We start with the conflicting clause $C_6 = \{\neg x_7, \neg x_8\}$ which is the obvious reason for the conflict because both its literals are set to $false$. As long as the literals are not assigned as decisions, we can further reason about why they are falsified. For example, we pick literal $\neg x_8$ and we look for a reason clause that falsified it (assigned x_8 to $true$). Looking back at the implication graph, we can see the reason is C_5. It means that setting other literals in C_5 ($\neg x_5$ and x_6) to $false$, led BCP to assign x_8 to $true$ which resulted in fasifying the conflicting clause C_6. So to prevent this conflict, we can not have all $\neg x_7$, $\neg x_5$ and x_6 to be $false$ at the same time. *AnalyzeConflict()* uses resolution to do this reasoning. By resolution on the conflicting clause, C_6, and the reason clause of x_8, C_5, as shown in figure 2.6, we can find a new clause C_7 as a reason for this conflict.

$$\frac{C_4{=}\{x_1,x_6,x_7\} \qquad C_7{=}\{\neg x_5,x_6,\neg x_7\}}{C_8{=}\{x_1,\neg x_5,x_6\}}$$

Figure 2.7: Further Resolution by CDCL to find an Asserting Clause

Remember that CDCL looks for an asserting clause which means it should only have one literal assigned at the current decision level. C_7 does not have this property as both x_6 and $\neg x_7$ belong to level 3 so the process will continue until an asserting clause is reached. Next, we pick x_7 (the last assigned variable) and resolve that by picking its reason clause which is C_4 (illustrated by the implication graph). Using resolution on C_4 and C_7 a new clause is derived called C_8 shown in figure 2.7. Note that only x_6 belongs to the current decision level, 3, so C_8 is an asserting clause and will be added to the clause database by CDCL as the learned clause.

In order to guarantee reaching an asserting clause, CDCL picks variables to resolve in the reverse chronological order of their assignment. So in the worst case, it tracks back to the decision made at the last level which is what happened in our example because we reached x_6 which was a decision variable. It is similar to backtracking and flipping the last decision, though the backtrack is usually to many levels before the last decision. We mentioned CDCL uses non-chronological backtracking to a level that only one literal of the learnt clause is unassigned at that level. For example in this example the literal in C_8 were assigned at 3 different levels. x_6 is from the last level 3, $\neg x_5$ is from level 2 and x_1 is from level 1. Non-chronological backtracking picks the highest level before the current level in the learned clause which is level 2 in this example and backtracks there. Since x_6 was not assigned at level 2, BCP immediately sets x_6 to $true$ to satisfy C_8 and we call it the *asserting literal*. Algorithm 4 illustrates this scheme where *AnalyzeConflict()* gets a conflicting clause $confl$ as an input and finds and reports a learned clause, C and a backtracking level, lev. Even though non-chronological backtracking is the main scheme used in most solvers, there

are some solvers with chronological backtracking in the recent years that are shown to be effective as well [83, 53] but we don't use them in this report.

Algorithm 4 Analyze Conflict

```
1: procedure AnalyzeConflict(confl)
2:     loop
3:         l ← Most recently assigned literal ∈ confl
4:         C′ ← reason(l)                          ▷ find reason clause for assigning l
5:         C ← resolution(C′, confl)
6:         if C is asserting then
7:             lev ← 2nd highest decision level in C
8:             return (lev, C)
9:         confl ← C
```

Modern SAT solvers stop once the first asserting clause is learned so this scheme is called *First Unique Implication Point learning* or *1-UIP learning* [78]. It is possible to continue the process and learn more clauses. For example if the asserting literal is not from a decision variable, we can still find a reason for that assignment and apply resolution further to learn a new clause. This process can be continued until the asserting literal is a decision (last-UIP) so potentially many clauses can be learned at each conflict. Learning 1-UIP clauses is shown to be the best learning scheme in multiple works [113, 38, 102]. This scheme guarantees to always learn a new clause at each conflict.

2.3 CDCL heuristics

Studying heuristics in algorithms goes back to 300 A.D in which Pappas suggested easy to use approximate methods that do not guarantee optimality [42]. Ever since, there are some general principles proposed to make an effective heuristic. For example, Bitner and Reingold were the first to propose one for the search rearrangement heuristic. It suggests that at every node of a search tree, the variable that has the smallest number of remaining choices (low degree) should be picked and assigned a value [25, 42].

Based on the nature of SAT algorithms, there are different heuristics proposed to help the solvers reach better performance. In the previous sections, we explained CDCL which is the core algorithm used in almost all modern SAT solvers. But there are many different heuristics added to this algorithm which makes solvers different from each other. These heuristics actually are shown to have a huge impact on the solver's performance. There are 3 main heuristics in each solver known as the Decision Heuristic, the Clause Database Management scheme and the Restart policy. Our focus in this work is on the first two which we will discuss in more detail later in this section. We first review the restart heuristic briefly.

The **restart** heuristic is a very simple but valuable technique. A restart is a backtrack to level zero where all variables are unassigned and the solver starts a new search. It was first introduced in [49] where it is argued that if a solver starts the search in a formula from a bad seed, there is a chance that it takes exponentially more time to solve it than from a good seed. This is known as *Heavy-Tailed* behaviour [48]. Restarting the search at random which is called *Randomized Restarts* results in dramatic improvements on satisfiable random formulas with heavy-tailed behavior [49]. Restarts were first aimed to try different seeds to increase the chance of finding a good seed that can solve the formula faster. For this argument to apply, there needs to be some randomness in the solver and that is why decision heuristics were initially randomized to make sure that after each restart the search tree would be different [95]. Other than that, solvers could not trigger restarts too frequently to make sure they can explore the complete search tree between two restarts in case the SAT problem they are trying to solve has no solution. This was no longer a necessity after CDCL algorithms came into the picture [77]. By learning and keeping new clauses, CDCL solvers can prove unsatisfiability even with rapid restarts [51]. Still, there is a lot of evidence showing restarts help solvers to have better performance on real word satisfiable and unsatisfiable formulas even when there is no randomness which is why it can be found in almost all modern solvers [68, 95, 15]. There are many different strategies in the literature for determining how often a restart should occur [73, 17, 8, 95].

In earlier CDCL solvers [46, 81, 82, 94], a fixed interval was used that triggers a restart after n conflicts where n has had different values between 550 as in Berkmin and 16000 as in Siege [46, 94]. It was suggested in [109] to use a geometric series for assigning values to n by starting with a small value first and the size of consecutive restarts grows geometrically. MiniSAT deployed this strategy and it was the first solver to show the effectiveness of this approach using the following geometric series: $n_i = 100 \times 1.5^{i-1}$ where i is the interval counter and n_i is the number of conflicts between $i-1$th and ith restart [39]. Other solvers like PicoSAT also showed improvements using this strategy [18]. A very popular geometric series used for assigning restart intervals in solvers is the Luby series [73] which is characterized by the following pattern: $1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, 1, ...$ In solvers RSat 2.0 [92] and TiniSat [54] the Luby series is used where each number in the series is multiplied by 512 and in PrecoSAT [19] and some more recent versions of MiniSAT [104] and their successors numbers are multiplied by 100. Local Restarts was presented in [95] with the intention of introducing a dynamic measure to restarts based on the current search of the solver and not just some fixed pattern. In this scheme a restart is triggered based on the decision level the search is at. It was observed that the clauses learned at lower decision levels are smaller hence more valuable. Therefore, giving less chance to higher levels forces the solver to prioritize learning smaller clauses [8, 95]. It has been shown that the local restart strategy can work well on solving unsatisfiable formulas when implemented in MiniSAT [95]. Another successful dynamic approach for restarts that was introduced in [10] and adopted

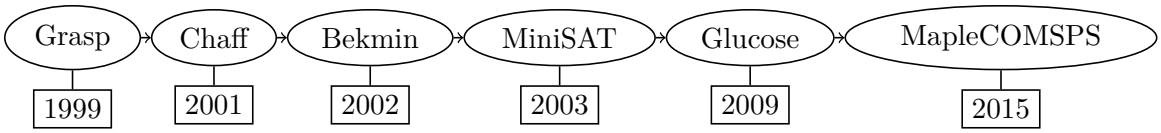


Figure 2.8: Evolution of Solvers

by many solvers after that [90], is based on the *LBD* of learned clauses which is a quality measure. Intuitively, the *LBD*-based restart heuristic, uses *LBD* to determine how good the clauses being learned by the solver are. If not good enough, a restart is triggered to start a new search. Compared to the previous restart heuristics, the *LBD* based restart heuristic as implemented in many solvers results in more rapid restarts in practice [10]. It is important that restarts happen along with clause learning to guarantee that the search after restarts is different. Note that adding new learned clauses to the solver prevents the same conflicts from happening. This guarantees the completeness of the algorithm with restarts.

As mentioned before, in addition to restarts, there are two main heuristics used in CDCL solvers. There is a lot of small details and hacks involved in the implementation of these heuristics and they may be used with various small changes in different solvers. Some solvers are specifically tuned to perform well on specific benchmarks. For example CryptoMiniSAT [101] that is specifically designed for solving cryptographic problems has heuristics tuned to work well on these benchmarks. Keeping this in mind, we picked a few solvers that were influential in the evolution of solvers and looked at their heuristics in more detail. They all made significant contributions to the current state of the art of solvers and we can find their footprints on most solvers used today. These solvers are **Grasp, Chaff, Berkmin, MiniSAT, Glucose** and **MapleCOMSPS**. Most of these were build up on the solver before or used many of their features. Figure 2.8 shows them in order of the year they were built in.

In the rest of this section, we explain the decision heuristic and clause database management and look at the implementations of them in these 6 solvers.

2.3.1 Decision Heuristics

In Algorithm 3 mentioned in last section, it is not clear how we choose the next variable to branch on. A very important heuristic in solvers, called decision heuristic (also known as branching heuristic), focuses on this matter. This heuristic is not specifically designed for CDCL and has been studied and used by SAT solvers and algorithms like DPLL before CDCL as well [76, 78, 42, 112, 35, 30, 60] One approach that called Variable State Independent Decaying Sum (VSIDS) [81], has been introduced in the Chaff solver and is probably the main decision heuristic of CDCL solvers ever since. Learning Rate Based (LRB) [69] is also a more recent decision heuristic that proved to be effective. In this section we look into

these two heuristics in more detail.

VSIDS: VSIDS was first introduced in the Chaff solver [81] and since then it has been the most known and used decision heuristic for CDCL solvers. It is known to be crucial for getting good performance on application benchmarks [23]. Intuitively speaking, the VSIDS heuristics tries to keep the search focused in a specific parts of the search space, so when choosing the next variable, it gives more priority to the ones that were involved in recent conflict analyses. In order to do that, VSIDS assigns an activity score to each variable or literal to represent how "active" they have been in previous conflict analyses. These activity scores are usually initialized with a constant value and increased by activity value v every time the variable appears in a learned clause so their chance of being picked as decision later is increased. At each round, the most active variable (variable with the highest activity score) is chosen by VSIDS as the next decision. VSIDS aims to give more weight to the more recently used variables and to do that, all activity scores are decayed over time so if a variable is inactive for a long time, its chance of being a decision is reduced compared to more recently active variables. In order to make sure the activity scores don't overflow, the scores are reduced uniformly once they reach a maximum value, max. so when the activity value of a variable is equal to or higher than max, the activity value of all variables are divided by max at once. Note that the original heuristic in chaff assigned activity values to each literal but starting from SATZOO and then MiniSAT, it was changed to variables [39].

LRB: A more recent decision heuristic called LRB was introduced in [69] and is used in many winning solvers in SAT competitions ever since [14, 53]. It looks at the solver's heuristic as an optimization problem and uses machine learning techniques to solve it. Particularly, LRB is based on a Multi Armed Bandit (MAB) algorithm [106], which is the problem of deciding what action to take at time t if the rewards of choosing all action so far are known in order to maximize the reward at time t. Viewing the SAT solver branching heuristic as a MAB problem, LRB defines choosing variables as actions and defines the Learning Rate (LR) measure for each variable as the rewards of choosing it as decision (actions in MAB) which needs to be optimized. LR measures how much a choosing a variable as a decision contributes to the solver making progress. As a measure of progress, it uses the production of learned clauses. First, it is said that a variable has participated in producing a learnt clause if it appears in that clause or it was resolved during the analysis of that clause. This means the variable was needed for getting a conflict and making a learnt clause (it appears somewhere in the implication graph when analyzing the conflict). Now to compute the LR of a variable v in interval I, let $L(I)$ be the total number of learnt clauses in I and $P(v, I)$ be the number of learnt clauses that v participated in during I. I is defined as the time that variable v is assigned until it becomes unassigned again by backtracking.

So $L(I)$ is the number of conflicts that occurred in I. Now we have $LR_v = P(v, I)/L(I)$. Since we can't compute the LR of a variable before the conflict is made and the analysis has happened, we use learning algorithms for a better estimation of LR. The greedy approach is to compute the mean of LR of each variable at all intervals so far and pick the one with the highest mean as the next decision. But since the solver is non stationary and the rewards change over time, an exponential moving average (EMA) [29] of each decision variable is used. So each variable v has a floating point number Q_v, which represents the EMA of all known LRs of it so far. Every time a variable v becomes unassigned, Q_v is updated by $Q_v = (1 - \alpha).Q_v + \alpha.LR_v$ where α (step size) emphasizes the importance of more recent values. α is initialized to 0.4 and gradually increased to 0.6 over time. Q_v is quite similar to activity scores of variables in VSIDS and is used to decide the next decision variable (variable with highest Q) by the solvers using LRB as their decision heuristic.

Phase Saving

Phase Saving also known as *Polarity Saving* is a technique that is shown to be important to be used as part of decision heuristics [91] and is usually implemented along with *VSIDS* and *LRB*. It is an approach for assigning the polarity when choosing a decision variable. It basically keeps the last known polarity of each variable, x, when it is assigned by BCP as P_x where P_x can be $true$ or $false$. If later, x becomes unassigned as a result of backtracking and is picked as a decision variable, the solver will set its polarity to P_x using phase saving approach. In [91], the authors observed that backjumps can result in repetitively solving the same satisfiable subformulas. Phase saving was proposed to prevent that from happening.

Chosen Solvers

Above, we explained the most well known decision heuristics that have been utilized by CDCL SAT solvers. It is important to note that each solver may adopt a different version of these heuristics, change some of the parameters or even introduce its own heuristic for picking the decision variable. It is beyond the scope of this report to explain them one by one as many new solvers are introduced every year. As mentioned before, we picked 6 solvers that played an important role in the evolution of modern SAT solvers and in the rest of this subsection, we will briefly go over the decision heuristics used in these 6 solvers.

GRASP [78]: GRASP was developed before VSIDS and uses a more straightforward decision heuristic. Its decision heuristic is called *Dynamic Largest Individual Sum (DLIS)*. Basically, it aims to satisfy as many clauses as possible with the decision assignment. When choosing the decision variable, it calculates the number clauses that would get satisfied by each assignment and picks the variable that satisfies most clauses. At each step, assuming

the number of unassigned variables is V, there will be $2 \times V$ cases to choose from, one for each literal (assigning variables to $true$ or $false$).

Chaff [81]: The Chaff solver introduced the Variable State Independent Decaying Sum (VSIDS) heuristic for picking the decision variables. VSIDS in chaff assigns two activity score to each variable (one for each literal) and chooses the one with highest activity value that is not currently assigned as decision. The polarity of the variable is also chosen by VSIDS as each literal has its own activity value and the picked literal determines the polarity of decision variable. Activity values are initialized to zero and every time a conflict happens, the activity values of the literals in the learned clause are incremented by 1. To keep the search more focused on recent learned clauses, all activity values are periodically divided by a constant which is called decaying. In Chaff, this constant is set to 2 and decaying happens after every 255 decisions. Literals are periodically sorted based on their activity values. Even though VSIDS computations add some overhead to the solver compared to decision heuristics used before, Chaff showed it can improve solver's performance by an order of magnitude.

Berkmin [46]: Berkmin's decision heuristic aims to be more dynamic and adopt to the current state of the search rapidly. Berkmin assigns an activity value to each variable instead of each literal as in Chaff. After each conflict, the activity values of variables appearing in the learned clause and also variables appearing in the clauses used in conflict analysis are increased by one. One main difference between Berkmin and Chaff is that Berkmin takes into account all clauses used in conflict analysis whereas in Chaff, only activities of the variables appearing in the learned clause are increased. All activity values are periodically divided by a constant which is 4 in Berkmin. This constant is 2 in chaff so BerKmin favors recent learned clauses more. The activity computation introduced by Berkmin was later adopted by zChaff [75, 44], an updated version of Chaff which was the winner of SAT 2002 and SAT 2004 competitions [66, 1]. Interestingly, activities are not the only or even the main measure Berkmin uses in its decision heuristic. Berkmin stores all learnt clauses chronologically in its database. To pick the decision variable, it finds the most recent learned clause that is not satisfied, called *current top clause*, and among its unassigned variables, it picks the one with highest activity as decision variable. If all learned clauses are satisfied, it acts like chaff and picks the variable with highest activity among all unassigned variables. So far we explained how Berkmin chooses a decision variable. After that, Berkmin uses two measures for assigning a polarity to the decision variable. The first measure is activity of literals shown by lit_act. Activity of a literal is calculated similar to the activity of variables except that there is no decaying. If the decision variable belongs to a learned clause (current top clause), this measure is used to set its polarity. For example assume v is the decision variable and l and $\neg l$ are literals made from v in positive and negative format respectively. Then if $lit\_act(l) > lit\_act(\neg l)$, v will be set to $true$ by Berkmin. If not, v will be set to

$false$. For setting polarity of other variables (in case all learned clauses are satisfied), a cost function, nb_{two} is used. For each literal l, function $nb_{two}(l)$ is computed as the number of binary clauses containing l plus the number of clauses containing $\neg l'$ where l' is the other literal appearing in these binary clauses. $nb_{two}(l)$ is considered as an estimate of the power of BCP after setting l to 0. Back to our previous example of decision variable v, and l and $\neg l$ being literals made from v, if $nb_{two}(l) > nb_{two}(\neg l)$, then v will be set to false. If not, v will be set to $true$.

MiniSAT [39]: MiniSAT introduced an efficient and popular implementation of VSIDS that is called Exponential VSIDS (EVSIDS) [39, 23]. This implementation of VSIDS has shown to be more efficient and has been used by other solvers ever since. EVSIDS assigns an activity score to each variable instead of the literals like Berkmin. At each round, the most active variable (variable with the highest activity score) is chosen as decision variable. In MiniSAT, these activity scores are increased by activity value v, every time a variable is either seen in the learnt clause or in clauses appearing in conflict analysis. Decaying is also different in this implementation of VSIDS. Instead of decaying the activity scores, the activity value (v) is increased over time so the activity score of more recent variables will end up higher. Lets assume v is initialized with 1 which is the case in original MiniSat solver so if a variable is used in the first conflict analysis, its activity score is increased by 1. In the next conflict analysis, the activity value will be increased by some $f > 1$. In MiniSat solver, $f = 1/0.95$ and $1/f$ (0.95 in this case) is called the variable decay factor [39]. Therefore, the activity scores of variables used in the second conflict analysis, will be increase by 1.05 instead of 1. In order to prevent overflow in activity scores, once one of them reaches a maximum value, max, all activity scores along with activity value v are uniformly divided by max. In MiniSAT max is set to 10^{100}.

Glucose [9]: Glucose uses EVSIDS as its decision heuristic and increases the activity score of variables used in conflict analysis by an activity value, v defined similar to MiniSAT activities. Glucose is known to introduce a clause quality measure, LBD, that will be explained later in Section 2.3.2, Clause Database Management. Glucose favors low LBD clauses and it increases the activity scores of variables involved in making such clauses more than other variables. So, it increases the activity value of all variables which were used in conflict analysis by v, and if they were propagated by a clause with LBD lower than the LBD of the new learned clause, they will be increased by $2v$ instead of v.

MapleCOMSPS [71]: This solver uses a combination of EVSIDS and LRB as it's decision heuristic. MapleCOMSPS starts with EVSIDS for the first 10000 conflicts as implemented in MiniSAT. Then it switches to LRB for 2500 seconds. If no answer is found in that time, the solver switches its decision heuristic back to EVSIDS and continues with that.

Note that in this solver, each variable has two different activity scores, one for each heuristic.

2.3.2 Clause Database Management

As mentioned before, modern SAT solvers learn a new clause after each conflict. They normally add thousands of clauses per second [87, 89]. These clauses are stored in the learned clause database and keeping all learned clauses can be impractical for two main reasons. First, keeping clauses consumes memory that can eventually result in exhaustion of the available memory of the computers used to run the solvers. Note that the number of new learnt clauses grows with the number of conflicts and this growth can be exponential in the number of variables of the formula [24]. Second, it is costly to perform BCP on clauses and not all clauses are valuable enough to be worth the cost. For example, large learned clauses (clauses with many literals in them) have shown to be less useful for search pruning purposes [99] and adding them to the database leads to additional overhead for conducting the search process. So, it eventually costs more in terms of BCP computations than what it saves in terms of backtracks [24].

Given that CDCL solvers generate a very large number of new learned clauses and based on the two reasons explained above, clause database management methods are central to solver's performance [6]. In particular, most learned clauses must be deleted to keep the clause database of practical size, and the clause reduction scheme is the key heuristic in the solvers to do that [9, 90]. There are two main aspects to a clause deletion strategy: The first is a method to categorize clauses as likely to be useful (high quality), or not (low quality). The better we can predict the future usefulness of clauses, the more efficient solvers we can have by removing less useful clauses and reducing BCP time. Second is implementation of an algorithmic method to remove low quality clauses efficiently. In an idealized scheme, having a clause quality measure Q, we can keep the clauses in a heap so that the lowest quality clause(s) can be removed when the clause database is deemed too large. Conventional wisdom is that using a heap would be too inefficient. It also seems unlikely that spending time to obtain the very worst clause is necessary. Thus, fast heuristics are desired.

In early CDCL solvers, some of the clauses were periodically deleted mainly to address the memory exhaustion problem [6, 81, 39, 46] but in the past 10 years clause database management became an active topic of research and has been shown to play an essential role in solvers' performance [9, 6, 11] so more well studied strategies has been introduced for it. From early solvers to current state of the art solvers, the fundamental way of managing learned clauses is to periodically detect some clauses with lower quality and delete them from the clause database. Determining the intervals for performing clause deletion is usually based on the number of conflicts since it is an indicator of the number of new clauses in the database. Note that after each conflict, a new clause is learned and added to the clause database. As discussed before, CDCL is a complete algorithm because of its learning

ability that prevents a conflict from happening twice which ensures eventually exploring the whole search space. This is only guaranteed if we keep all learnt clauses and not with clause deletion. In order to preserve completeness of the algorithms, some solvers gradually increase the intervals between clause deletions. Given that the number of variables in each formula is finite, the number of possible learned clauses made from those variables is finite as well so if the interval between two clause deletions is long enough for the solver to learn all possible clauses, completeness is ensured. Some solvers use restarts as a point to perform clause deletion and do it right after a restart [46].

The scheme that is widely used in solvers, which we call *Delete-Half*, is to periodically sort the learned clauses based on some quality measure and delete the half with lowest quality. We will look into some of the main clause quality measure in the next section.

Clause Quality Measures

The quality measure is typically a combination of size, age, literal block distance (LBD) and some measure of usage or activity [9, 39, 69, 90, 89]. First, lets look at each of these measures and explain how each has been used as a quality measure.

Size: Size of a clause is the number of literals in it. It is broadly accepted that the smaller clauses are more valuable and have higher quality for different reasons. They have fewer literals so they need less memory to be kept. A clause of size n is visited by BCP at most $n-1$ times so the smaller the clause, the less time spent by BCP on it at worst case. Also, since it is likely that a smaller clause becomes unit under some assumption faster than a larger clause, it can enable more unit propagation and result in finding a conflict faster.

Age: Age of a clause is determined by how many conflicts were made since a clause was learned. It is believed that the more recent clauses are more relevant to the current search and so more valuable (higher quality).

LBD: *Literal Block Distance* or *LBD* was first introduced in [9] as a measure of learned clauses quality and is computed to reflect the number of decision levels involved in a clause, assuming that every literal in the clause is assigned. Consider a learned clause $C = \{x_1, x_2, x_3\}$. Assume that at the time the clause was learned, x_1 was assigned at level 10, x_2 was assigned at level 20 and x_3 was the asserting literal. Since there are 2 decision levels involved in this clause, the *LBD* of the clause is set to 2. *LBD* of a clause is at most equal to the size of the clause minus 1. It has been shown that the lower the *LBD* of a clause, the more useful the clause is in future hence the higher its quality [9].

Activity and Usage: This measure is mainly used to reflect how useful a clause has been so far. The Usage is measured by the number of conflicts for which the clause has been responsible for so far (used in conflict analysis). Activity or VSIDS Activity is a similar measure that takes the time of usage into account. Basically, the clauses that were used more recently are considered to be more usefull and have higher activities. VSIDS strategy was first introduced in Chaff solver for variables [81] but it was later adopted by MiniSAT to

be used for clauses as well [39]. Clause activity is calculated similar to the variable activity. Every time a clause is used in conflict analysis, its activity is increased by a value. This value increases over time to give more weight to more recent usages.

Chosen Solvers

Now, lets get to the clause database management schemes implemented in our chosen solvers. In all solvers clauses that are a reason for a current variable assignment are saved from deletion. These are the clauses that have all literals except one assigned false and the remaining literal has been assigned true by BCP.

GRASP [78]: GRASP uses size as its clause quality measure. It aims to keep the size of the learned clause database to be polynomial in the number of variables. GRASP defines a threshold, k, and only stores the clauses of size k or less in the database and keeps larger clauses around only while they are unit. As mentioned before, all clauses are considered to be unit at the time they are learned with respect to the partial assignment of that time (only their asserting variable is unassigned) and this will change when a backtrack happens and other variable/variables in the clause that where assigned at a higher level than the backtrack level become unassigned as well. The clauses of size larger that k are deleted then. k is set to 20 in the original GRASP solver.

Chaff [81]: Chaff uses size and assigned variables in the learned clauses to choose which clauses should be deleted and when. When a clause is learned, it is examined to determine when (if at all) it should be deleted. For each clause a threshold t is set and when the number of unassigned variables in the clause reaches t, it is marked as "deleted" and the solver will ignore it from that point on. Periodically, the actual memory assigned to "deleted" clauses is freed. t is usually between 100 and 200 in the solver.

Berkmin [46]: Berkmin performs clause deletion after each restart using a combination of age, size and usage as quality measures. Their hypothesis is that since more recent clauses were made in higher decision levels they are harder to derive and so more valuable to keep. (Note that after each restart the decision level resets to zero and increases over time). They consider the 15/16 of the clauses that were learned more recently to be "new" and the other 1/16 to be "old". From the old clauses any clause with size larger than 8 or usage smaller than 60 will be removed at each restart. From the new clauses any clause with size larger than 42 or usage smaller than 7 will be removed. The usage threshold value of the old clauses is increased after every 1024 decisions with the starting value of 60. So if a large clause is not used in conflicts and does not have an increasing usage will be removed from the database eventually. Berkmin aims to delete at least 1/16 of clauses when a clause deletion is activated. If after applying the above scheme, the number of removed clauses are less that 1/16 of all learned clauses, the size threshold value of old clauses is decreased by one (starting at 8). However, this threshold will never be less than 4 and based on experimental

results shown in Berkmin's paper, this is enough to always delete 1/16 of clauses on their benchmark.

MiniSAT [39]: MiniSAT uses a combination of size and activity in its clause deletion method. It utilizes the *Delete-Half* scheme and periodically deletes half of the learned clauses in database based on these measures. The learned clauses are in a list and when a deletion is triggered, all binary clauses will go to the beginning of the list and the rest of the clauses will be sorted based on their activity in a descending order. Then, half of the clauses from the end of this list are deleted. MiniSAT also deletes all clauses of size larger than 2 with activity smaller than a threshold even if they appear in the beginning of the list. This threshold is set to 1 over the number of learned clauses. The deletion is triggered after every i conflicts where the value of i increases over time to allow more learned clauses in the database. In early versions of MiniSAT i is initialized to the number of clauses in the original formula divided by 3. After each restart, i is multiplied by 1.1 to gradually increase the size of clause database.

Glucose [9]: Glucose is mainly known for introducing LBD as a clause quality measure. Ever since, LBD has been used in different heuristics of state of the art solvers. Glucose is also took a more aggressive approach to learned clause deletion and showed that storing fewer clauses in the database could improve a solver's performance. Like MiniSAT, Glucose also uses the *Delete-Half* scheme to reduce the size of its clause database but sorts the learned clauses based on their LBD and deletes the half with larger LBDs. It activates the deletion more frequently so generally maintains a smaller database. To be more specific, no matter the size of the initial formula, Glucose removes half of the learned clauses with lower LBDs every $20000 + 500x$ conflicts where x is the number of times clauses deletion was previously performed. The main advantage of aggressive clause deletion is faster BCP which is shown in particular to help on satisfiable instances [8].

MapleCOMSPS [71]: MapleCOMSPS adopted the deletion scheme introduced in another solver called COMiniSatPS [90, 89]. It partitions the clause database into three different sets, called Core, Tier2 and Local. The decision of where to store a newly learned clause in is based on its LBD at the time. A clause is stored in Core if its LBD ≤ 3, in Tier2 if $4 \leq$ LBD ≤ 6 and in Local if $6 <$ LBD. After the first 100,000 conflicts, if there are not enough clauses in Core (less that 100), the core threshold is changed from 3 to 5. A clause may be moved from one set to another based on LBD or usage. The LBD of each clause is recomputed whenever it is used in conflict analysis. If the LBD of a clause at that time is sufficiently reduced to meets the thresholds, it will be moved from Local to Tier2 or Core, or from Tier2 to Core. Based on its updated LBD, every 10,000 conflicts, all clauses in Tier2 that have not been used during the last 30,000 conflicts are moved to Local. The deletion only happens to clauses stored in Local. After every 15,000 conflicts, all the clauses in Local are sorted by their activity and half of the clauses with lower activities will be removed from the database. Clauses with recent improvement in LBD are saved from deletion.

2.4 Summary

The Boolean Satisfiability Problem (SAT) is to determine whether a given boolean formula can be made to evaluate to true by assigning boolean truth values to the variables in the formula. SAT is a canonical decision problem shown to be NP-complete and is one of several computational tasks identified by researchers as core problems in computer science [90]. Due to its NP-complete nature, there is no algorithm that can solve SAT problems in polynomial time unless we prove P=NP, which is believed to be very unlikely. In the worst case, a general SAT solver based on search principles needs to visit an exponential number of nodes in the search tree to determine the satisfiability of a Boolean formula. Thanks to the efficiency of modern SAT solvers, which is probably their most important feature, we are far away from this worst case scenario in solving real world SAT instances.

A lot of the research done in the SAT community is about improving the speed of SAT solvers and this has been done by trying to reduce the number of nodes that are visited during the search by improving the algorithms and heuristics used in solvers. For example, learning, a good decision strategy and non-chronological backtracking have been studied and shown to be efficient by empirical experiments.

In this work, we looked at CDCL algorithm which is widely used as the main algorithm in SAT solvers and has been known to be one of the main reason for efficiency of solvers. We also discussed the most important heuristics used in the CDCL algorithm which had a very important role in the evolution of modern SAT solvers. To illustrate this evolution, we focused on 6 important solvers and their heuristics. These solvers have had an impact on the solvers coming after them as they all introduced new strategies and demonstrated how they can improve the performance. Table 2.4 summarizes the main features of each solver. I believe there is a lot of room for improving SAT solvers efficiency further by studying these current methods and their impact on different types of real world formulas.

GRASP	• First CDCL solver that illustrated the efficiency of CDCL algorithm • Uses DLIS (Dynamic Largest Individual Sum) as decision heuristic • Deletes clauses based on their Size • No Restarts
Chaff	• Placed 1st in main track of SAT competition 2002 and 2004 • Introduced VSIDS decision heuristic • Deletes clauses based on their Size • Uses a fixed-size restart strategy
Berkmin	• Placed 1st in main track of SAT competition 2002 (Satisfiable instances) • Used a modified version of VSIDS as decision heuristic which is calculated for all variables and is updated in all used clauses in conflict analysis • Deletes clauses based on their Size, Age and Usage • Uses a fixed-size restart strategy
MiniSAT	• Placed 2nd in main track of SAT competition 2005 and 3rd in SAT competition 2007 • Introduced EVSIDS decision heuristic which is a more efficient implementation of VSIDS • Deletes clauses based on their Size and Activity • Uses a geometric series restart strategy
Glucose	• Placed 1st in main track of SAT competition 2011 (All instances), 2012 (All instances) and 2013 (Unsatisfiable instances) • Placed 2nd in main track of SAT competition 2009 (All instances) and 2014 (Unsatisfiable instances) • Uses EVSIDS as decision heuristic with LBD-based modifications • Deletes clauses based on their LBD • Uses a LBD-based restart strategy
MapleCOMSPS	• Placed 1st in main track of SAT competition 2016 • Uses a combination of LRB and EVSIDS as decision heuristic • Deletes clauses based on their LBD and Activity • Have 3 tiers for saving learned clauses • Uses a combination of LBD-based and Luby restart strategies

Table 2.4: Main Features of the 6 Chosen Solvers

Chapter 3

Decision Heuristics and Instance Structure

3.1 Overview

Structure seems important in SAT research in attempts to explain the large divergence in CDCL solvers performance over formulas of various sorts. For example, large industrial formulas are often easier than small crafted or random formulas. Notions of instance structure are invoked in explaining solver performance and many empirical studies relate aspects of solver performance to formula structure. Modularity is an appealing notion of useful structure. If a formula can be decomposed into sub-formulas with few shared variables, then each assignment to the shared variables induces smaller sub-problems that can be solved independently [7]. An example of such an approach is the use of tree decompositions. Unfortunately, while industrial formulas have interesting structure, few have suitable decompositions where the shared or "connecting" set of variables is small enough for standard decomposition methods to be useful in practice [80]. Also, it is not at all clear how to exploit such decomposition approaches in a CDCL solver for enumeration of all assignments to be effective. Our approach has been to search for lightweight methods for exploiting structure within the scheme of current high-performance CDCL SAT solvers.

Structure of CNF formulas is often studied in terms of an underlying graph, such as the primal graph. The "community structure" of primal graphs of industrial CNF formulas is one of the structural properties that have received some attention in the literature in recent years [5, 87, 4, 70]. These studies have shown interesting correlations between community structure of CNF formulas and aspects of the execution of CDCL SAT solvers on those formulas. In the notion of community structure, a graph is considered to have "good community structure" if its vertices can be partitioned into "communities" so that the ratio of the number of within-community edges to the number of between-community edges is greater than would be expected in a random graph with the same degree distribution. This can be thought of as a kind of modularity, and indeed one measure of quality of community structure, introduced

in [86], is named modularity in the literature. It has been observed that this modularity measure is correlated with CDCL SAT solver running times [87]. Variables appear in clauses that belong to two or more communities are called "bridge" variables. It has been observed that the VSIDS decision heuristic in CDCL SAT solvers chooses bridge variables much more frequently than other variables [70]. This raises several questions. What causes the VSIDS preference for bridge variables? Is this preference good or bad? Are there other structural properties that are similarly relevant to VSIDS preference? Can we manipulate this preference to alter solver behaviour? Our results in this chapter of the thesis partially answer the latter three questions.

Towards answering these and related questions we carried out a variety of experiments. First, we measured the degree to which VSIDS prefers several "special" families of variables. Second, we proposed different ways to utilize these variables in modifying a CDCL solver's heuristics to improve their performance. We started with a simpler solver, Glucose, and then showed our methods have benefits for other state of the art solvers as well. We chose Glucose because it is well known and relatively well understood, is easy to work with, and is very influential in that it is the base of several other solvers.

Community graphs of industrial formulas are diverse in structure, but we observed that some have or contain a rather simple, nearly linear, structure (see Section 3.4). For these formulas, the variables in the central communities of this structure can be viewed as "connecting variables" between two sub-formulas. This led us to examining the most central variables in the formulas, based on a global measure called betweenness centrality [43]. We found that VSIDS also shows a strong preference for the high-centrality variables. Bridge variables and high centrality variables appear to be very interesting. However, the sets of these variables are too large to consider enumerating all assignments to them. We determined that, rather than doing this, we can simply encourage VSIDS to choose these variables more often. In VSIDS based solvers, solvers make their decisions based on the "activity" of variables and these activities are determined by a "bumping" method. When a clause is learned, a set of variables that were effective in the learning process are chosen and their activities will be increased by a factor called the "bump value". The bump value is usually set to 1 and increases at regular intervals to give more weight to recent activities. We can simply encourage VSIDS to choose a special set of variable more often by adjusting/increasing the "bump value" for that set of variables. We call this "preferential bumping". We show here that preferential bumping of high centrality and bridge variables can be used to improve the performance of high-performance CDCL SAT solvers. In our initial experiments we showed improvements when modifying Glucose with preferential bumping on these variables.

Observing the impressive effect of utilizing variable betweenness centrality to improve Glucose performance, we also present modifications to the decision heuristics of a state-of-the-art solvers. We give three different centrality-based modifications that alter VSIDS and LRB variable activities. We demonstrate the effectiveness of the methods by implementing

them in Maple LCM Dist, the winning solver from Main Track (Industrial track) of the 2017 SAT solver competition, and running them on the formulas from that track with a 5000 second time-out. All the modifications increased the number of instances solved and reduced the PAR-2 scores. While our methods are simple, to our knowledge this is the first time that explicit structural information has been successfully used to improve the current state-of-the-art CDCL solvers on the main track benchmark. We also report a number of other measures of solver performance and make some observations about these. We show that:

- VSIDS has strong preference for a number of "special" families of variables; Bridge variables, high degree variables and high centrality variables.
- Sufficient structural information to improve performance with "preferential bumping" on these variables can be obtained quickly enough to be useful.
- By selectively altering the VSIDS "bump value" for specific sets of variables, we can influence the preference VSIDS has for choosing these variables.
- Modifying VSIDS and LRB decision heuristics to prefer high centrality variables can improve the performance of a state of the art CDCL solver.
- The effect of our modifications is positive on a few different performance measures other than solving time.

3.2 Related Work

Several papers have studied the structure of industrial CNF formulas, e.g., [9, 110, 47]. Stronger "Community structure" has been shown in industrial formulas [41] and this community structure quality is correlated with solver run time [4, 5, 87, 88]. Community structure was used in [6] to generate useful learned clauses in satisfiable instances. They added a preprocessing step in which, the solver tries to solve the subformulas made from the clauses in pair of communities. For the satisfiable subformulas, all learned clauses will be added to the original SAT formula. After all possible learned clauses are augmented to the original formula, the solver will start solving this new formula instead. They showed this can improve solver's performance on satisfiable instances. In [79, 85] community structure is used in SAT-based MaxSAT solvers. There, structure is used to partition the formula to obtain smaller witnesses of unsatisfiability, not to direct the SAT solver execution. [70] showed that VSIDS tends to choose bridge variables (community connectors) as decisions more often. [100] described a method that alters the parallel portfolio solver PeneLoPe [2] to heavily focus on specific communities at each time. Community structure is also used in [108] to improve performance of parallel solvers by defining high quality clauses based on LBD and number of communities they appear in. Hierarchical Community Structure (HCS)

is another definition of community structure proposed in [67] to explain the efficiency of SAT solvers in industrial formulas. It has been shown HCS correlates well with solver's run time and is an effective measure to distinguish between easy industrial and hard crafted SAT instances. Eigenvalue centrality of variables was studied in [61], and it was shown that CDCL decisions are likely to be central variables. Some features used in learning performance prediction models, as used in SATzilla [111], are structural measures. Lower bounds for CDCL run times on unsatisfiable formulas are implied by resolution lower bounds, and formula structure is central to these [16]. Formulas with bounded treewidth are fixed parameter tractable [3], and also can be efficiently refuted by CDCL with suitable heuristics [7].

3.3 Structural Properties

In this section, we define some structural measures of CNF formulas. We also show that, although the time to compute these measures for some formulas is prohibitive, for the majority of formulas it is fast enough to be used as a pre-processing step in a SAT solver. We assume the reader is familiar with the standard terminology regarding graphs.

Primal Graph

The primal graph of ϕ (also know as variable incidence graph or the variable interaction graph (VIG)) is usually used to study the structure of CNF formulas. The nodes of the primal graph are the variables of the formula, and there is an edge between two nodes if the corresponding variables co-occur in a clause (in either polarity). Let ϕ be a CNF formula with m clauses over n atoms (Boolean variables) in S. The frequency of atom p in ϕ is the number of occurrences of p, negated or not. The primal graph of ϕ is the graph $G(\phi) = \langle V, E\rangle$ with $V = S$, and $(p, q) \in E$ iff there is a clause $C \in \phi$ containing both p and q either negated or not. By the degree of atom p we mean the degree of p in $G(\phi)$. Degree of an atom p also indicates the number of clauses that p appears in.

Community Structure

The community structure of a formula is typically based on a weighted version of this graph, where each clause of size k contributes weight $1/\binom{k}{2}$ to each associated edge. If G is a weighted graph and P a partition of its vertices, then define:

$$Q(G, P) = \sum_{p \in P} \left[\frac{\sum_{w,y \in p} w(x, y)}{\sum_{w,y \in V} w(x, y)} - \left(\frac{\sum_{x \in p} deg(x)}{sum_{x \in V} deg(x)} \right)^2 \right]$$

The modularity Q of G is the maximum value of $Q(G, P)$ over all partitions P. We say a formula has good community structure if the modularity Q of its weighted primal graph is

large. Computing a partition that maximizes Q is NP-hard, so for large graphs, heuristic methods are used. As in most previous SAT research on community structure, we use the Louvain method [26], as implemented in the NetworkX python package, to compute the community structure of formulas [72] in this chapter. Edges with end points in distinct communities are called bridges, and nodes in such edges are called bridge variables. Bridge variables are the "connecting" variables in this notion of modularity. Clearly, bridge variables appear in clauses that belong to two or more communities. A community graph for a formula has communities as vertices and an edge between two communities if there is a bridge between them. Weights may be used to reflect community size and the number of bridges between two communities. It has been shown that VSIDS decision heuristic shows a preference for bridge variables [70].

Centrality

The intuitive notion of "centrality" of a vertex in a graph may be measured in a number of ways. A very simple centrality measure is equal to vertex degree and is sometimes called degree centrality. The Eigenvalue centrality of a vertex v is the value $a[v]$ where a is a normalization of the eigenvector of the adjacency matrix of G having maximum eigenvalue [27, 61]. Our focus in this chapter will be on the Betweenness centrality.

The betweenness centrality of a vertex v in a graph G is the number of shortest paths between pairs of vertices excluding v, that visit v. It is defined by $g(v)$ as follows, where $\sigma_{s,t}$ is the number of shortest s-t paths and $\sigma_{s,t}(v)$ is the number of those that pass through v, normalized by the number of all possible paths to range over $[0, 1]$ [43].

$$g(v) = \sum_{s \neq v \neq t} (\sigma_{s,t}(v)/\sigma_{s,t})$$

The betweenness centrality of a variable v in formula ϕ is the betweenness centrality of v in the primal graph $G(\phi)$. Exactly computing betweenness centrality involves an all-pairs-shortest-paths computation, and is too expensive for large formulas. We computed approximate centrality values using the NetworkX [84] betweenness_centrality function, with sample size parameter $n/50$, where n is the number of variables. The function call is as follows, where the parameter sets the number of vertices used to compute the approximation:

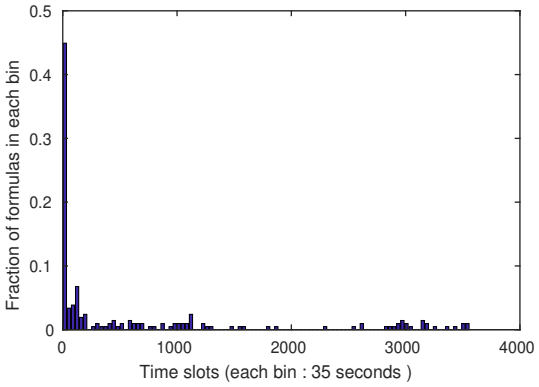
$$nx.betweenness\_centrality(G, k = samplesize, normalized = True)$$

All computations of the experiments reported in sections 3.3 and 3.4 were run on Intel(R) Core(TM) 3.4 GHz i7-3770 quad-core processors (single-threaded) with 16Gb of RAM, running Linux.

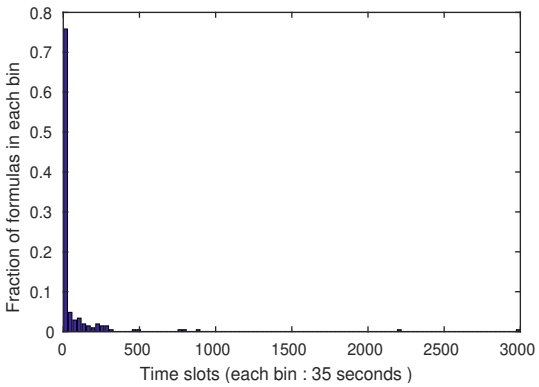
3.3.1 Structural Properties Computations

For a significant fraction of formulas from industrial benchmarks, the time to detect community structure or compute variable centralities is substantial, and often greater than the time to decide satisfiability. For our exploration of the cost of computing structural measures and their role in CDCL solvers, we used a set of formulas consisting of all 600 formulas from the industrial categories of the 2013 and 2016 SAT solver competitions for which we could perform centrality computations within one hour, using sample size set to the number of variables divided by 50. The resulting set has 223 formulas from a wide variety families within the competition benchmark sets. The community detection took less than one hour for each of these formulas. We used this set of formulas in the experiments reported in this section and in Section 3.4.

For many industrial formulas we could not obtain good approximations of centralities in less than an hour. Clearly, one could not (at least with current methods) perform these computations in a SAT solver for every input. However, for a large fraction of the formulas reasonable approximations can be computed quite quickly. Figure 3.1a shows a histogram of the approximate centralities we could find in one hour. As can be seen, for many formulas the computation is quite fast, and only a small fraction have relatively long computation times. More than half these formulas required less than 70 seconds. Figure 3.1b shows a histogram of community detection times, which shows a similar pattern. The community detection generally was faster than centrality computation, and more than three quarters of the formulas required less than 35 seconds.



(a) Histogram of Centrality Computation Times.



(b) Histogram of Community Detection Times

Figure 3.1: Computing time of structural properties

Further, it turned out that the most useful information for our preferential bumping schemes (at least as currently implemented) that will be explained later was generally obtained quickly. For example, for all instances that were solved by Glucose enhanced with centrality-based preferential bumping (GLPB-HC-i as described in Section 3.4.2) but not

solved by default Glucose with the same time-out (see Section 3.4.4), centrality computation time was less than 150 seconds.

It seems slightly counter-intuitive that useful information for hard formulas should be easy to compute. We examined the correlation between running time of default Glucose, and the times to compute centrality and community information. Table 3.1 shows the results. Because SAT solver run times are very far from normally distributed, it is more appropriate to use a non-parametric measure of correlation than, for example, the well-known Pearson correlation. We use the Kendall τ rank-based correlation coefficient. A τ of 1 indicates perfect correlation, -1 perfect inverse correlation, and 0 no correlation at all. As can be seen, the times to compute both structural measures are highly correlated with each other. Glucose solving time is only weakly correlated with structure computation times. However, the p values for these measures are extremely small ($p < 0.01$ is generally regarded as highly significant), so we can be fairly confident this correlation is real. The interesting point is that these correlations are negative which suggests computing structural information is easier for formulas that are harder for Glucose. Of course, for similar formulas of different size, structure computation time must increase with formula size. The negative correlation of solver time and structure computation time may reflect a bias in the benchmark selection process: The small formulas are hard relative to their size, and the large formulas are easy relative to their size.

Measures	Kendal τ	p-value
Glucose *vs.* Centrality computation	-0.13	0.003
Glucose *vs.* Community detection	-0.20	$\ll 0.0001$
Centrality computation *vs.* Community detection	0.71	$\ll 0.0001$

Table 3.1: Correlations amoung structure computation and solver times.

3.4 VSIDS Preferences and Preferential Bumping

As previously discussed, the VSIDS decision heuristic, with a number of small variations, has been the dominant decision heuristic almost since it was introduced in [81]. A useful recent discussion of VSIDS can be found in [23]. The VSIDS method is based on recording an "activity score" for each variable of the formula. At each decision, the unassigned variable with highest score is chosen. As implemented in Glucose, the activity score of a variable is updated each time the variable is seen while deriving a new learned clause. The update is performed by adding the current "bump value" to the score. The bump value is initially 1, and at each conflict it is multiplied by $1/d$, where d is the "decay factor" (VSIDS approximates an exponential moving average), which is initially 0.8 and is gradually increased to 0.95.

3.4.1 VSIDS Preferences

It was observed in [70] that VSIDS chooses bridge variables with far greater frequency than might be expected. Here, we confirm that this holds for our test benchmark set, and add several more observations. We were interested in the degree to which VSIDS, in Glucose, would show a preference for high degree variables, for high centrality variables, and also for high degree and high centrality bridge variables. In [70] it is claimed that VSIDS prefers high centrality bridge variables. The centrality measure used there is degree centrality, a local property, whereas the betweenness centrality measure we use here is a global property.

We ran Glucose on the formulas of our test benchmark set, and recorded the number of decisions from each variable family. Table 3.2 shows the mean value of each of these measures over the 186 formulas from our test benchmark set that Glucose solved within the 5000 second time-out.

Family	Atoms (%)	Decisions (%)	Strength	Hit Rate	Precision
All	100	100	1.0	100	37
Bridge	49	83	2.86	77	64
High Degree	33	80	2.38	62	50
High Centrality	33	73	2.22	45	52
High Deg. Bridge	22	67	4.05	49	71
High Cent. Bridge	21	65	4.17	36	82
High Deg. High Cent.	18	65	3.8	34	69

Table 3.2: Measures of the degree to which VSIDS prefers "special" families of variables.

The first column of Table 3.2 identifies the family of variables in question. We defined the set of high degree variables to be the one-third fraction of variables with the highest degree, and high centrality similarly. The second column gives the fraction of all atoms in the formula that belong to each family. High Degree Bridge, High Centrality Bridge and High Degree High Centrality are the intersection of variables in two different families. The third column of Table 3.2 gives the fraction of all decisions that are variables from each family. We observe that, for each family, the fraction of decisions belonging to the family is much higher than the fraction of atoms belonging to that family, indicating that VISDS chooses variables from these structurally "special" families more often than others. In fact (see discussion of the Precision column below) on average 63% of variables are not chosen as decisions even once during a run of the solver.

The forth column is the ratio of the fraction of decisions from each family to the fraction of atoms from the family. It is the mean of the ratios, not the ratio of the means, so can't be computed from the preceding columns. This gives a measure of how "important" the family is relative to its size. The sets of bridge, high degree and high centrality variables all account for about twice as many decisions as their sizes would suggest. The high degree

bridge and high centrality bridge sets, while much smaller, account for far more decisions relative to their size hence have higher values in the forth column.

The fifth and sixth column are standard evaluations of the predictive power of a factor, where we are viewing membership in the special family as a predictor of being chosen at least once as a decision variable. Hit rate, also called sensitivity or "true positive rate", is the fraction of special atoms that are chosen by VSIDS at least once during a run. Precision, also called "positive predictive value", is the fraction of variables that are chosen at least once that are also from the family. Notice that the All row of the Precision column gives us the fact that only 37% of variables are chosen as decisions at some time during a run.

Our data show that, while VSIDS preferentially chooses several families of "special" variables, some of these preferences are much stronger than others, and in particular, some are much stronger relative to the size of the "special" set. Although the high degree bridges and high centrality bridges seem very interesting, in preliminary experiments, preferential bumping of these sets did not appear especially promising. It is not clear whether this is because there are just too few of them or something else.

VSIDS Preferences are Stable

Our sets of "special variables" are static, being defined in terms of the initial formula. However, It has been observed in [6], and confirmed by our own experiments, that clause learning destroys the initial community structure of formulas. That is, the primal graph made from the set of clauses the solver is working with after running for a while does not have the same community structure as the initial formula. In particular, a large fraction of learned clauses would add bridges to the initial set of communities.

Since VSIDS, overall, prefers variables with certain initial structural roles, one might suspect that this preference changes during a run as the structure of the current clause set changes. Since our sets of "special variables" are computed only on the initial formula, it might be expected that they become less relevant over time. We cannot determine if that is the case, but we did find, despite the changing learned clause set, the preferences VSIDS shows for the "special" families of interest here are generally quite stable of an entire run of the solver.

Figure 3.2 is illustrative of several experiments we ran that show this. The figure shows the average fraction of decisions from three families of special variables over a run. To show full-run data and also the initial "settling down", we use an unconventional x-axis. The left half of the axis shows the values every 100K decisions for the first 1 million decisions. The right half shows the values at 9 points equally spread over the remainder of each run.

3.4.2 Preferential Bumping

The strong preference shown by VSIDS for certain families of variables led us to wonder whether these preferences are always good or not, and to consider whether we could alter

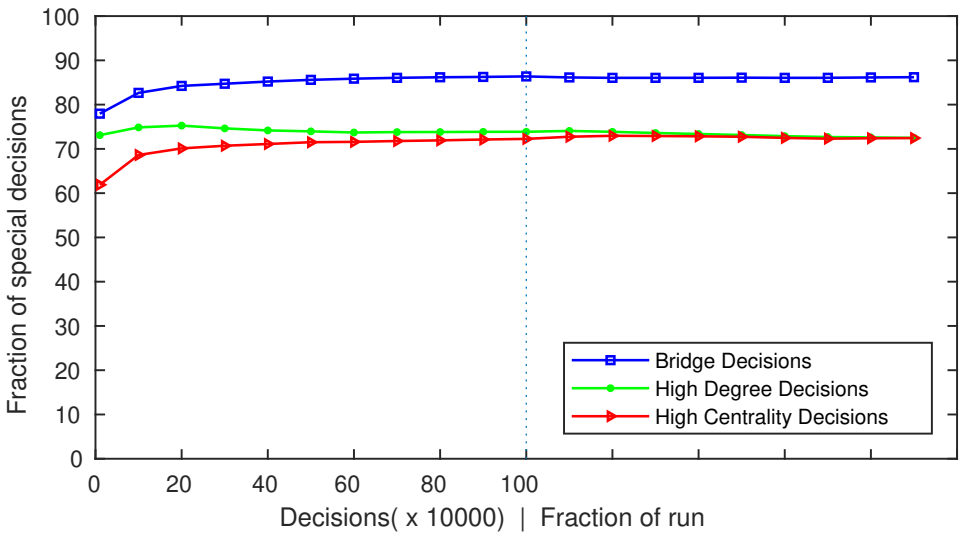


Figure 3.2: Mean fraction of special decisions over full run. Left half is for the first 1M decisions; right half is the remainder of the run to completion.

the fractions of these families of decisions, thereby perhaps adjusting for "non-optimal preference". We found that by selectively increasing or decreasing the VSIDS activity bump value for certain variables, we can increase or decrease (respectively) the tendency of VSIDS to pick them. We illustrate this here by showing the fraction of Glucose decisions that are from a chosen family of variables when we increase the bump value for the variables in that family of variables. We experimented with a number of preferential bumping schemes, two of which we report here.

Our preferential bumping scheme alters the VSIDS score update factor, "bump value", for a selected set S of variables. For variables not in S, the score update is addition of the default bump value, as described above. We maintain a second "special" bump value, which is initialized to some value b different from 1, and is updated in the same manner as the default bump value. This "special bump value" is used to update the scores of variables in S. We verified that, for a number of special families of variables, increasing the special bump value, increases the fraction of decisions among the special family, while decreasing the special bump value reduces the fraction of decisions among the special variables.

The two preferential bumping schemes we report data on below are as follows. Having selected a set S of variables we want to influence, and a "special bump value", we do one of:

1. Uniform preferential bumping scheme: Variables from S are bumped using the "special" bump value during the entire run;

2. Initial preferential bumping scheme: Variables from S are bumped by the "special" bump value until 100K decisions have been made, after which they are bumped using the default bump value.

We call this Glucose modified with preferential bumping GLPB, and use names of the form GLPB-S-B, where S identifies the set of special variables and B identifies the preferential bumping scheme. S is either HC, for "high centrality variables" or Br for "bridge variables". B is either u for "uniform preferential bumping" or i for "initial preferential bumping". In all the reported experiments with Glucose, the special bump value is initiated to 1.1 instead of 1.

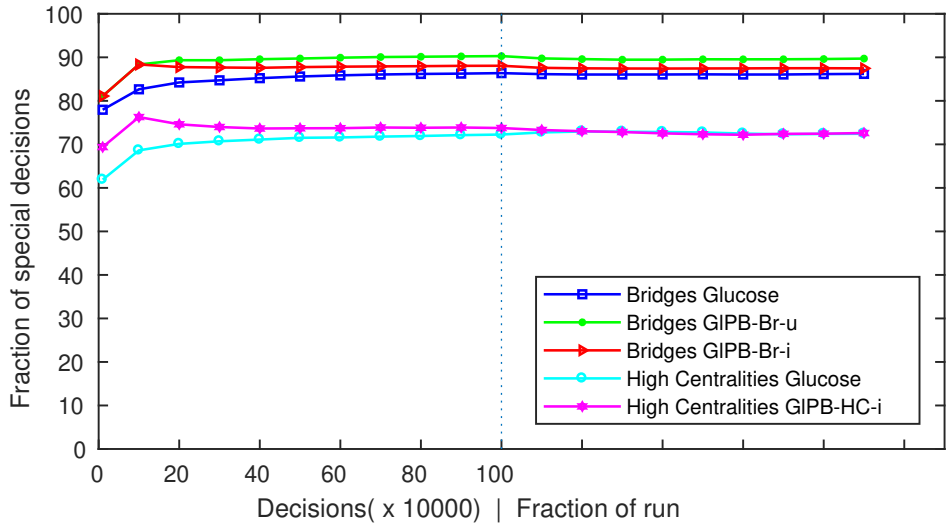


Figure 3.3: Effect of preferential bumping on the fraction of special decisions.

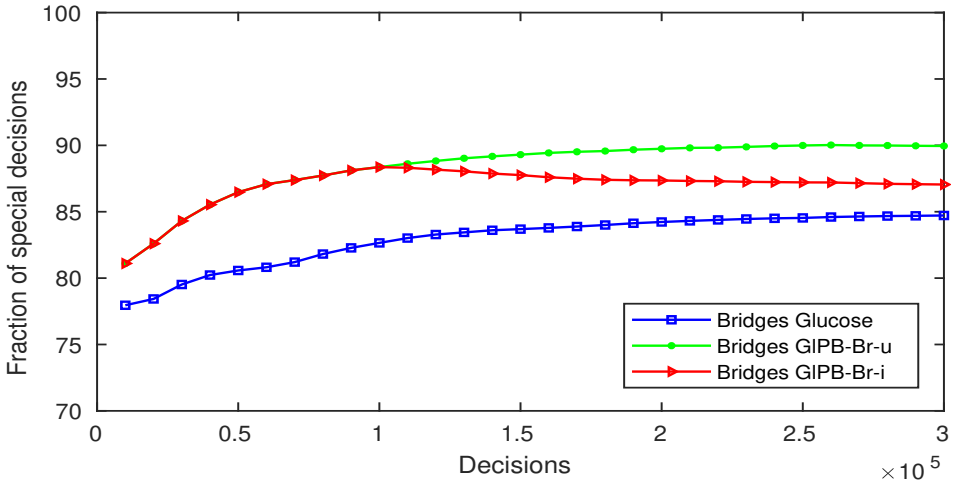


Figure 3.4: Effect of initial *vs* uniform preferential bumping: magnified inital segmant.

Figure 3.3 illustrates the effect of preferential bumping. The two lower lines are fractions of high centrality decisions for default Glucose and for the variant GLPB-HC-i, with initial preferential bumping of high-centrality variables. The three upper lines show the fraction of bridge decisions for default Glucose and for the solvers GLPB-Br-u, with preferential bumping of bridges all the time, and GLPB-Br-i, with initial preferential bumping of bridges.

Figure 3.4 shows a blow-up view of the initial segments of those three curves. Preferential bumping for the entire run increases the fraction of "preferred" decisions for the entire run. With initial preferential bumping, the fraction increases at first, then drops back toward, but does not meet, the default value.

Now that we described the preferential bumping, in section 3.4.4 we will demonstrate the effectiveness of structure-based preferential bumping in two versions of GLPB (Glucose modified with preferential bumping), on a large set of industrial formulas. We begin with a small motivating example in section 3.4.3.

3.4.3 Preferential Bumping of Central Communities

Community graphs of many industrial formulas do not seem to always have an easy-to-understand structure but we observed that they do in many other cases. Figure 3.5 gives examples of the former; Figures 3.6 examples of the latter. This is not surprising to see these structures in industrial formulas since they are modeling real worlds problems versus the random formulas. In particular, figures 3.6 shows these have an imperfect but clear linear "coarse structure".

In these figures, the size of nodes reflects community size (number of variables in each community), and the width of edges reflects the number of bridges between the relevant communities (number of variables shared in between 2 communities). The formula UR-15-10p0 takes the form of a linear chain-like structure. The formula hwmcc10-timeframe-expansion-k45 takes the form of a path of large communities connected by wide edges, plus a number of small communities connected by narrow edges. We view the dominant linear structures here as being a kind of coarse structure. It is the apparent structure if we ignore most small details. There are many other formulas with "nice coarse structure" – another example is shown in Figure 1 of [6].

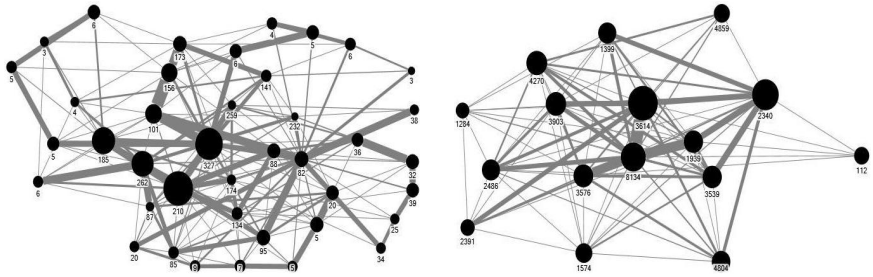


Figure 3.5: The not-so-nice community graphs of formulas minxorminand128 (left) and AProVE07-03.

To examine the effect of increasing bump value of central variables we start with an illustrating experiment. For a number of formulas with a clear linear coarse structure, we selected a small number of (visually) central communities, and then increased the Glucose bump value for the variables in these communities. For the formulas hwmcc10-timeframe-

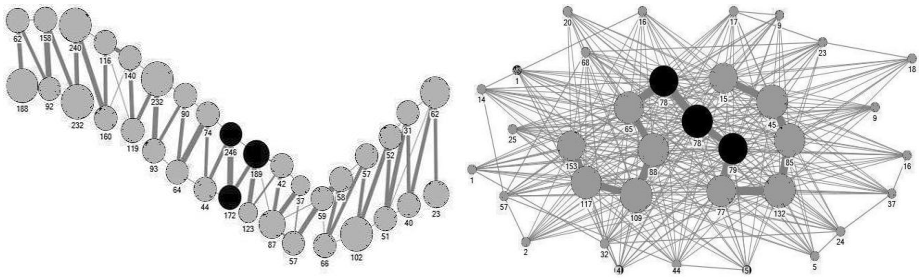


Figure 3.6: The community graphs of instances hwmcc10-timeframe-expansion-k45 (left) and UR-15-10p0 (right) have a linear "coarse structure". Black nodes are those selected for preferential bumping. (Numeric labels are number of decisions, in thousands, made by Glucose with preferential bumping, within each community.)

expansion-k45 (which we write as hwmc..k45 for short) and UR-15-10p0, the selected communities are the highlighted in black in Figure 3.6.

Instance	Default Glucose		Glucose + Central Bumping	
	Central Decisions	CPU Time	Central Decisions	CPU Time
hwmc...k45	8.5 %	2147	12.6 %	616
UR-15-10p0.cnf	6.3 %	1293	15 %	749

Table 3.3: Effect of increasing the bump value for variables in central communities on solving time and fraction of decisions

Table 3.3 shows, for each of the formulas of Figures 3.6, the effect of uniform preferential bumping on the fraction of decisions within the chosen central communities, and the corresponding reduction in Glucose run times, which is significant. Our selection of central communities in these examples was done manually, but since the community graphs of most formulas are small, we can quickly compute good path decompositions (and probably useful tree or branch decompositions) of them, so finding central communities in community graphs like these can certainly be automated.

These, and similar examples, provided some of the first evidence of success of structure-based preferential bumping, and motivated exploration of the notion of centrality in SAT formulas' primal graphs.

3.4.4 Preferential Bumping in Glucose

In the following, we demonstrate that fully automatic preferential bumping of specific families of structurally determined variables can in fact improve solver performance. In particular, we ran default Glucose, GLPB-HC-i and GLPB-Br-u on all of the 742 distinct formulas from the industrial categories of the three most recent (at time of experiments) SAT solver competitions (2013, 2014 and 2016) combined. This set includes the 223 formulas used in previous sections of this paper.

Recall that GLPB-Br-u preferentially bumps variables that are bridges in the primal graph for the entire run, and GLPB-HC-i preferentially bumps variables that have high centrality in the primal graph for the first 100K decisions. The computations to find bridge and high centrality variables can be expensive, in some cases taking longer than running a SAT solver on the instance. However, for many formulas, the computations are fast. Moreover, it seems that the information obtained tends to be more useful when the computation is fast than when it is not. Therefore, the strategy of simply running bridge detection or centrality computation algorithms for a short time as a pre-processing step is useful. If we obtain the structural information in time, we use it, otherwise we simply run the default version of the solver and pay only a small time penalty. We allowed up to 50 seconds for community bridge detection in GLPB-Br-u and 200 seconds for centrality computation in GLPB-HC-i, and 5000 seconds total running time (structure computation plus solving) for each formula. 5000 seconds is the standard timeout for the applicational track of SAT competitions [1]. If the structural properties computation was not successful within the time out, GLPB-HC-i and GLPB-Br-u continue running as the default glucose solver for the remaining time.

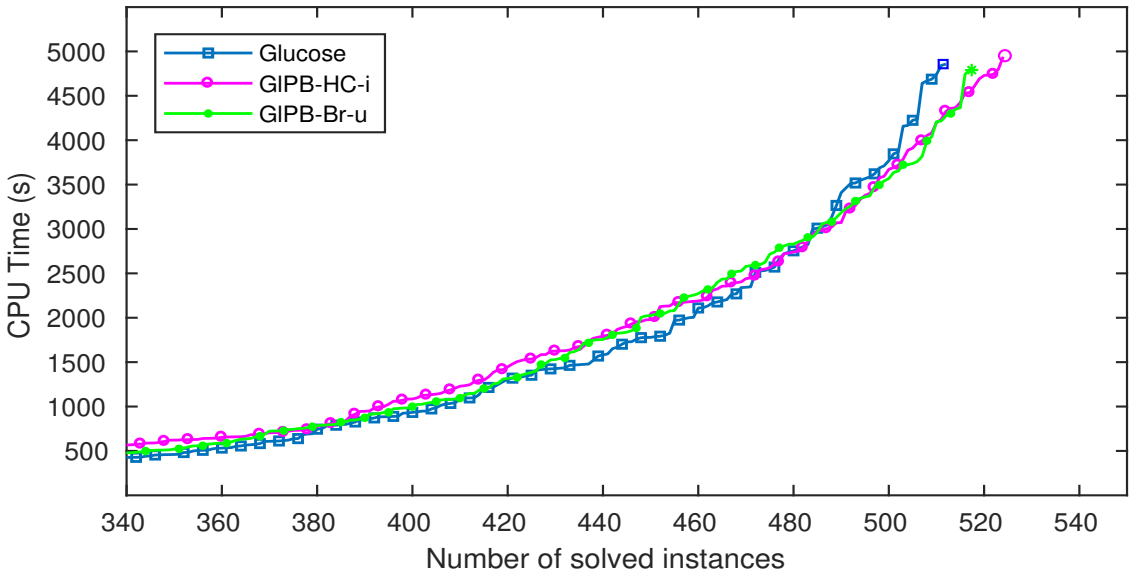


Figure 3.7: Relative performance of Glucose, GLPB-Br-u and GLPB-HC-i.

From the 742 industrial formulas in the benchmark set, 210 were not solved by any of the three solvers. Glucose solved 512 formulas, GLPB-Br-u solved 517, and GLPB-HC-i solved 524. For calibration, in the 2016 competition 4 formulas is the difference between first and third place, and in the 2020 competition 4 formulas is the difference between first and second place. GLPB-HC-i also solved one formula from the 2016 SAT solver competition that none of the 29 solvers in the competition solved within the competition timeout. While our time-out is the same as the competition, our hardware is not, so we cannot conclude

that GLPB-HC-i would have solved it in the competition. However, the fact that no solver in the competition solved it in 5000 seconds does suggest that it is indeed a hard formula by current standards.

Figure 3.7 is the standard (in the SAT solver literature) "cactus plot", which shows for each solver how many formulas were solved in time at most t, as a function of t. We see that on easier formulas the time spent on structure computation does not pay off, and Glucose performs best. However, the preferential bumping solvers perform better on harder formulas.

Robustness of Results

Solver	Number Solved (out of 60)	Mean solving time
Glucose	54	347
Altered Glucose	54	355
GLPB-HC-i-0.9	54	382
GLPB-HC-i-1.1	56	353
GLPB-HC-a-0.9	55	507
GLPB-HC-a-1.1	55	430
GLPB-Br-i-0.9	54	398
GLPB-Br-i-1.1	54	349
GLPB-Br-a-0.9	54	422
GLPB-Br-a-1.1	56	381

Table 3.4: Comparing performance of modified solvers on a smaller benchmark. The mean solving time is over the formulas that all solvers could solve.

Alterations to SAT solvers can be tricky to evaluate, because even a very small change can significantly alter the performance on any individual formula. This leads to a number of ways in which apparent but not real improvements are seen. For example, if one chooses a set of formulas which are relatively hard for a specific solver, relative to other good solvers, it is often the case that almost any small change at all will improve the performance *on those formulas*. Such changes typically lose on other formulas in a good benchmark set.

In our experiments we, unsurprisingly, found many settings in which we seemed to get improved performance. The methods we have reported are ones for which we have seen enough data to have reasonable confidence are not just a lucky coincidence. Table 3.4 gives one such data set. This is for runs on a smaller benchmark set, with a number of parameter settings. The data shows that when we bump bridges harder with any of our schemes, we get better performance, and when we bump them less we get worse performance. Similarly for high centrality. The line "Glucose" is for completely unmodified Glucose, while the line "Glucose Altered" is Glucose with our modifications, but all parameter changes turned off. It is slightly slower because it contains code which is always executed, but does no work when our modifications are turned off.

3.5 Centrality based Modifications in Maple LCM Dist

In the previous experiments, we illustrated a few ways to utilize the structural properties of the SAT formulas to improve the performance of a basic CDCL solver, Glucose. Those are interesting findings but it is imprortant to us to see if they are also beneficial to the state of the art solvers with more complicated heuristics. For that, we chose the MapleSAT family of solvers, which have been the winners of the SAT competition between 2015-2019 [1]. They all share similar decision heuristic which is a combination of VSIDS and LRB. In the rest of this section, our focus will be on modifying these two decision heuristics in Maple-LCM-Dist solver, the winner of SAT competition 2017, and utilizing betweenness centrality of the variables in the formulas to improve its performance.

3.5.1 Decision Heuristics

The VSIDS decision heuristic [81], in several variations, has been dominant in CDCL solvers for well over a decade. Recently, the LRB (Learning Rate Based) heuristic [69] was shown to be effective, and winners of recent competitions from MapleSAT solvers family use a combination of VSIDS and LRB. As discussed previously, both employ variable "activity" values, which are updated frequently to reflect the recent variable usage. Similar to VSIDS, the update in LRB involves increasing the activity value for a variable each time it is assigned or appears during the derivation of a new learned clause. A secondary update in MapleSAT [69] and its descendants involves, at each conflict, *reducing* the LRB activity score of each variable that has not been assigned a value since the last restart. Maple LCM Dist uses both VSIDS and LRB, at different times during a run, and LRB activity reduction.

In section 3.4.4, we reported that increasing the VSIDS bump value for high-centrality variables during an initial period of a run improved the performance of the solver Glucose. This did not help much in solvers using LRB, but motivated further studies to have LRB favor high-centrality variables. The modifications reported here are:

HCbump-V: We scale the VSIDS additive bump values for high-centrality variables by a factor greater than 1. In the experiments reported here, the factor is 1.15. This is similar to VSIDS preferential bumping described in section 3.4.2 with uniform bumping scheme.

HCbump-L: We periodically scale the LRB activity values of high-centrality variables by a factor greater than 1. In the experiments reported here, we scaled by a factor of 1.2 every 20,000 conflicts.

HCnoReduce: We disable the reduction of LRB scores for "unused variables" that are also high-centrality variables.

3.5.2 Performance Evaluation

We implemented each of our centrality-based decision heuristics in Maple LCM Dist [74], the solver that took first place in the Main Track of the 2017 SAT Solver Competition [96]. We compared the performance of the modified versions against the default version of Maple LCM Dist by running them on the 350 formulas from the Main Track of the 2017 solver competition, using a 5000 second time-out. This is the standard time-out in SAT competitions [1]. Computations were performed on the Cedar compute cluster [31] operated by Compute Canada [33]. The cluster consists of 32-core, 128 GB nodes with Intel "Broadwell" CPUs running at 2.1Ghz.

We allocated 70 seconds to approximate the variable centralities, based on the cost-benefit trade-off seen in Figure 3.1a: Additional time to obtain centralities for more formulas grows quickly after this point. If the computation completed, we used the resulting approximation in our modified solver. Otherwise we terminated the computation and ran default Maple LCM Dist. The choice of 70 seconds is not crucial: Any cut-off between 45 and 300 seconds gives essentially the same outcome. Centrality values were obtained for

	Maple LCM Dist	HCbump-L	HCbump-V	HCnoReduce
Number Solved	215	219	218	221
PAR-2 Score	4421	4382	4375	4381

Table 3.5: Number of Formulas Solved (out of 350) and PAR-2 score, for default Maple LCM Dist and our three modified versions.

198 of the 350 formulas. Our 5000 second timeout includes the time spent on centrality computation, whether or not the computation succeeded.

Table 3.5 gives the number of instances solved and the PAR-2 score for each method. All three centrality-based modifications improved the performance of Maple LCM Dist by both measures.

Figure 3.8 gives the "cactus plot" (inverse cumulative distribution function) for the runs. All three modifications result in improved performance. these solvers, which modify the decision heuristic, improve on the default for all times longer than 3300 seconds. The two methods that modify LRB under-perform the default on easy formulas, but catch up at around 3200 seconds.

Families Affected

It is natural to wonder if these improvements are due to only one or two formula families. They are not. Table 3.6 shows, for each of our three modified solvers, how many formulas it solved that default Maple LCM Dist did not, and how many families they came from.

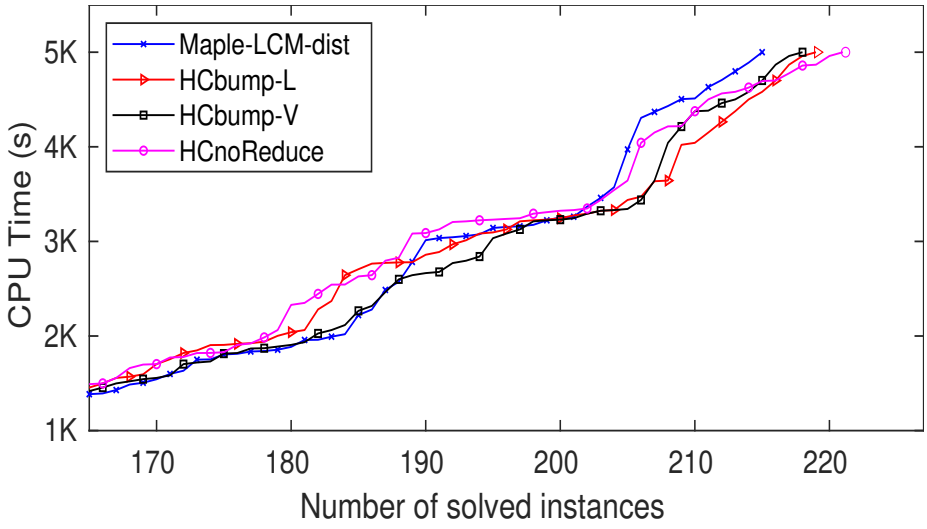


Figure 3.8: Cactus plot comparing performance of Maple LCM Dist and our three modified versions.

Table 3.6 shows that the improvements happens in formulas from various families and is not specific to a special domain.

Solver	HCnoReduce	HCbump-L	HCbumpt-V
Number of Formulas	10	8	5
Number of Families	6	5	3

Table 3.6: Number of families involved in formulas solved by our modified solvers by not by default Maple LCM Dist.

3.5.3 Performance Analysis

In this section, we compare the performance of our modified solvers in more details.

Reliability

There is an element of happenstance when using a cut-off time. For example in figure 3.8, the “best" method would be different with a cut-off of 2800 seconds, and the “worst" would be different with a cut-off of 3500 seconds. Run-time scatter plots give us an alternate view.

Figure 3.9 gives scatter plots comparing the individual formula run-times for each of our three modified solvers with the default Maple LCM Dist. Each point in the scatter plots represents a formula. The satisfiable (SAT) formulas are denoted with o and the unsatiafiable (UNSAT) formulas are denoted with $\times$. The x-value and y-value of each point shows the solving time of that formula by 2 solvers. If a formula was not solved within the time-out of 5000 seconds, the solving time value is replaced by 5000.
We observe:

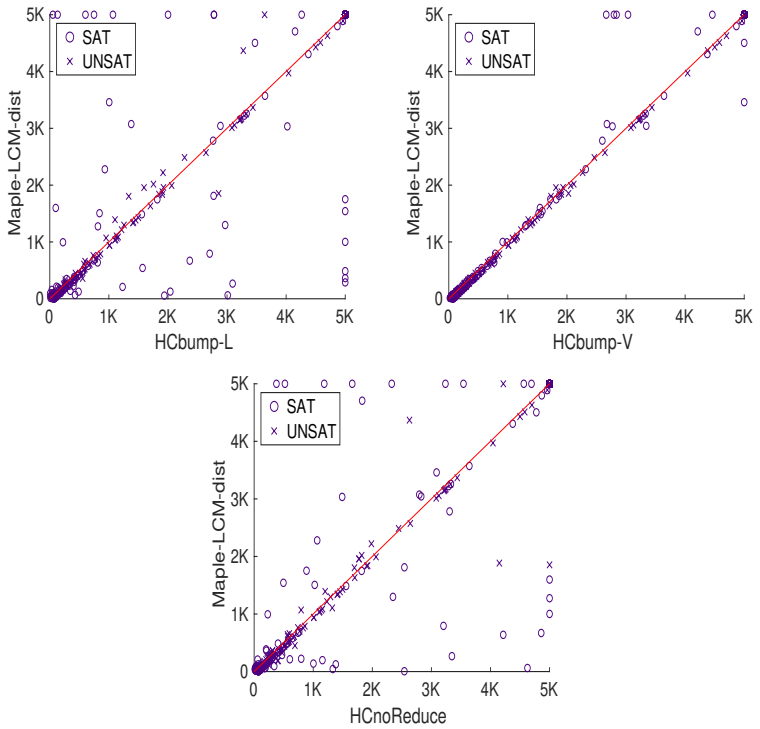


Figure 3.9: Comparison of run-times of default Maple LCM Dist with each of our modified versions. Each plot shows all instances that were solved by at least one of the two solvers represented. Satisfiable formulas are denoted with ○, unsatisfiable formulas with ×.

- In each plot many points are lined up just below the main diagonal. These are the formulas that we could not find the centralities for in time. For these, we pay a 70-second penalty.
- The VSIDS modification, HCbump-V, caused the least variation: it solved more formulas, but gave significant speedups on only a few others.
- The two LRB modifications, HCbump-L and HCnoReduce, were very "noisy", speeding up many formulas but also slowing down quite a few.
- It is very interesting that very large differences in run-time were mostly for satisfiable formulas.

Reasoning Rates

Here, we look at other performance measures to compare the solvers based on the rate of solver's reasoning or search (as distinct from, for example, quality of reasoning or total time). Table 3.7 shows, for each method, the mean conflict production rate, the mean decision rate and the mean unit propagation rate. These are the average number of conflicts, decisions and unit propagations made by the solver per second respectively over the benchmark set. The table also reports the average Global Learning Rate (GLR). GLR, was introduced in

[72] where it is defined as the ratio of the number of conflicts to the number of decisions in solving a formula. It was observed that solvers with decision heuristics that produce a higher GLR had reduced solving times [72]. We observe:

- Consistent with the observations in [72], decision heuristic changes that improved performance increased GLR, though only slightly;
- Modifying the LRB decision heuristic in HCbump-L solver increased all the search and reasoning rates and learned clauses faster than the default solver. This is not the case for the other two modified solvers.

Solver	Conflicts	Decisions	Propagations	GLR
Maple LCM Dist	8.25	23.7	1,452	0.623
HCbump-L	8.52	26.3	1,530	0.626
HCbump-V	8.15	23.5	1,432	0.625
HCnoReduce	8.02	21.0	1,420	0.629

Table 3.7: Measures of search or reasoning rate for the four solvers. Conflicts, Decisions and Propagations are in thousands of events per second.

3.6 Summary

We demonstrated that the community graphs of some formulas have nice "coarse structure" that can be exploited fairly easily. We have extended previous observations that the VSIDS decision heuristic in CDCL solvers shows a strong preference for particular families of variables. We also introduced the notion of variable betweenness centrality in SAT formulas and showed high centrality variables are among those families. Wondering whether this preference was good or bad led us to experiment with preferential VSIDS bumping. We have shown that the performance of Glucose can be meaningfully improved by preferential bumping schemes.

Glucose uses simple heuristics compared to the recent state of the art solvers in SAT competitions. To evaluate the effect of utilizing centrality in these solvers, we introduced three centrality-based modifications to standard CDCL decision heuristics, and implemented these in Maple LCM Dist, first-place solver from the Main Track (aka industrial track) of the 2017 SAT Solver Competition. All three changes improved the performance on the formulas from this track.

The decision heuristic modifications confirmed the importance of variable centrality, and are interesting because they seem to work for different formulas. For example, among 26 formulas that at least one method solved and at least one did not, there are 12 formulas that are either solved by HCbump-L and no other method, or not solved by HCbump-L but solved by all other methods.

Chapter 4

Clause Centrality in Deletion Strategies

4.1 Overview

In our attempt to develop lightweight methods for exploiting formula structure, we introduced the notion of centrality in SAT formulas in Chapter 3 and showed that it can be utlized in the decision heuristics of CDCL solvers to improve their performance. In this chapter, we introduce clause centrality as a measure of clause quality, and study its use in the clause database management heuristic.

We demonstrate the effectiveness centrality as a clause quality measure by implementing by showing that replacing the activity based deletion in Maple LCM Dist, the winning solver from Main Track (applicational track) of the 2017 SAT Solver competition, with centrality based deletion improved performance. We also report a number of other measures of solver performance and learned clause quality, and make some observations about these. We continue by moving the centrality computations as a preprocessing step done by the solver and confirm our improvements using another solver, MapleLCMDistChronoBT, the winner of the main track of SAT competition 2018.

To explain the good performance of solvers with centrality-based deletion, we look at the usefullness of clauses with high centrality in conflict analysis and how it compares with other quality measures. Finally, we compare various clause quality measures in the deletion scheme of MapleLCMDistChronoBT and illustrate the effectiveness of our measure. While our methods are simple, to our knowledge this is the first time that explicit structural information has been successfully used in clause database management heuristic to improve the current state-of-the-art CDCL solvers on the main track benchmark. We show that:

- Centrality of variables can be utilized in the clause deletion scheme of state of the art solvers to improve performance.

- A new clause quality measure can be introduced based on centrality of variables in the clauses in a way that clauses with high centrality values, are used more often in conflict analysis.
- The effect of our modifications is positive on a few different performance measures in addition to solving time.
- Small clauses, clauses with small LBD and High-centrality clauses are used more often in conflict analysis.
- As long as the solvers have a permanent set to store low LBD clauses, Centrality is the best measure to be used in the Delete-Half scheme based on our experiments.

4.2 Clause Centrality

In the previous chapter, we used betweenness centrality of variables in decision heuristics and we have evidence suggesting using centrality of variables in decision heuristics can help performance. Recall that in Chapter 3 we defined the betweenness centrality of a variable by $g(v)$ as follows, where $\sigma_{s,t}$ is the number of shortest s-t paths and $\sigma_{s,t}(v)$ is the number of those that pass through v, normalized by the number of all possible paths to range over $[0, 1]$ [43].

$$g(v) = \sum_{s \neq v \neq t} (\sigma_{s,t}(v)/\sigma_{s,t})$$

We start by introducing a clause quality measure, which we call clause centrality, to help use betweenness centrality of variables of a CNF formula in clause deletion strategies. In this measure, we consider the clauses with high centrality variables in them to have higher clause centrality values.

Clause Centrality: Centrality of a clause is computed as the mean betweenness centrality of the variables occurring in it. For a clause of size m like $C = \{v_1, v_2, ..., v_m\}$, the centrality is computed as follows:

$$Centrality(C) = \frac{\sum_{i=1}^{m} Centrality(v_i)}{m}$$

4.3 Centrality based Clause Deletion in Maple LCM Dist

The Maple LCM Dist clause database management (or clause deletion) scheme was inherited from COMiniSatPS which was explained in Chapter 2. A summary of the scheme is as follows [13, 74]. The learned clauses are partitioned into three sets called CORE, TIER2 and LOCAL. Two LBD threshold values, t_1, t_2, are used. Learned clauses with LBD less than t_1 are put in CORE. t_1 is initially 3 but changes to 5 if CORE has fewer than 100 clauses after 100,000 conflicts. Clauses with LBD between t_1 and $t_2 = 6$ are put in TIER2.

	Maple LCM Dist	HCdel
Number Solved	215	224
PAR-2 Score	4421	4242

Table 4.1: Number of Formulas Solved (out of 350) and PAR-2 score, for default Maple LCM Dist and HCdel solvers.

Clauses with LBD more than t_2 are put in LOCAL. Clauses in TIER2 that are not used for a long time are moved to LOCAL. For clause deletion, the clauses in LOCAL are ordered by non-decreasing activity values. If m is the number of clauses in LOCAL, the solver deletes the first $m/2$ clauses that are not reasons for the current assignment.

The clause deletion of Maple LCM Dist is basically the Delete-Half scheme in LOCAL using activities. We would like to see the result of changing the clause quality measure in deletion to centrality. We report on the following modification to Maple LCM Dist using centrality in its clause deletion scheme:

HCdel: Replace criteria for ordering the clauses in LOCAL from clause activity with ordering by clause centrality. In every clause deletion step, delete half the clauses with lower centrality values excluding clauses that are reason for the current assignment.

4.3.1 Performance Evaluation

We implemented centrality-based deletion, HCdel, in Maple LCM Dist [74], and compared the performance of the modified version against the default version of Maple LCM Dist by running them on the 350 formulas from the Main Track of the 2017 solver competition, using a 5000 second time-out. Computations were performed on the Cedar compute cluster [31] operated by Compute Canada [33]. The cluster consists of 32-core, 128 GB nodes with Intel "Broadwell" CPUs running at 2.1Ghz.

Exactly computing betweenness centrality involves an all-pairs-shortest-paths computation, and can be too expensive for large formulas. In the experiments of the first section of this chapter, section 4.2 and this section, we computed approximate centrality values using the NetworkX [84] betweenness_centrality function, with sample size parameter $n/50$, where n is the number of variables in the formulas.

We allocated 70 seconds to approximate the variable centralities, similar to experiments in chapter 3. Additional time to obtain centralities for more formulas grows quickly after this point but the choice of 70 seconds is not crucial: Any cut-off between 45 and 300 seconds gives essentially the same outcome. If the computation was completed, we used the resulting approximation in our modified solver. Otherwise we terminated the computation and ran default Maple LCM Dist. Centrality values were obtained for 198 of the 350 formulas. Our 5000 second timeout includes the time spent on centrality computation, whether or not the computation succeeded.

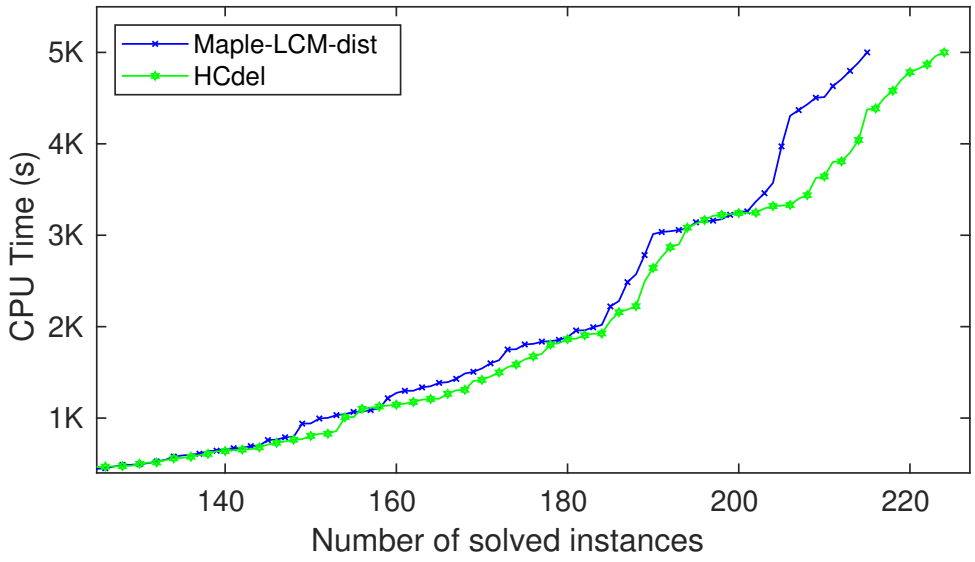


Figure 4.1: Cactus plot comparing performance of Maple LCM Dist and HCdel solvers.

Table 4.1 gives the number of instances solved and the PAR-2 score for both solvers. The centrality-based clause deletion scheme, HCdel, improved the performance of Maple LCM Dist by both measures.

Figure 4.1 gives the "cactus plot" for the runs. HCdel out-performs default Maple LCM Dist for almost all smaller cut-off time values. Within the standard 5000 seconds, HCdel solves 9 formulas more than the default solver which comes from 5 different family of formulas. It is interesting that the improvement are not specific to one or two families and could improve different kinds of instances.

Reliability

There is an element of happenstance when using a cut-off time as in cactus plots so it is useful to compare performance in other ways. Run-time scatter plots give us an alternate view.

Figure 4.2 gives a scatter plot comparing the individual formula run-times for both solvers. Each point in the scatter plots represents a formula. The satisfiable (SAT) formulas are denoted with o and the unsatiafiable (UNSAT) formulas are denoted with $\times$. The x-value and y-value of the point shows the solving time of that formula by the two solvers. If a formula was not solved within the time-out of 5000 seconds, the solving time value is replaced by 5000.

We observe that many points are lined up just below the main diagonal. These are the formulas without centralities, for which we pay a 70-second penalty. We can also see that HCdel was faster on 70% of formulas with centralities and had significant slow-down for only 4 formulas. It is interesting that very large differences in run-time were mostly for

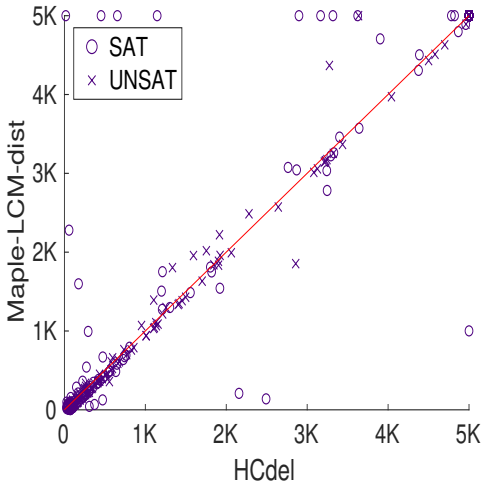


Figure 4.2: Comparison of run-times of default Maple LCM Dist with HCdel. The plot shows all instances that were solved by at least one of the two solvers. Satisfiable formulas are denoted with ∘, unsatisfiable formulas with ×.

satisfiable formulas. We observed the same effect before from solvers with centrality-based decision heuristics.

Reasoning Rates

Similar to analysing centrality-based decision heuristics, here we look at a few performance measures to compare the solvers based on the rate of reasoning or search. Table 4.2 shows, for each method, the mean conflict production rate, the mean decision rate and the mean unit propagation rate. These are the average number of conflicts, decisions and unit propagations made by the solver per second respectively over the benchmark set. The table also reports the average Global Learning Rate (GLR). GLR, was introduced in [72] where it is defined as the ratio of the number of conflicts to the number of decisions in solving a formula.

We observe that, HCdel did not have a higher GLR than Maple LCM Dist suggesting that it learned or kept "better" clauses after decisions and propagations, rather than more clauses. Looking at other reasoning rates, HCdel has higher reasoning rates in all measures suggesting it was generally faster in reasoning like generating conflicts.

Solver	Conflicts	Decisions	Propagations	GLR
Maple LCM Dist	8.25	23.7	1,452	0.623
HCdel	8.43	25.2	1,493	0.623

Table 4.2: Measures of search or reasoning rate for the solvers. Conflicts, Decisions and Propagations are in thousands of events per second.

4.3.2 Comparing Clauses after Deletions

Measures of "clause quality" that have been studied or used in solver heuristics include size, literal block distance (LBD) and activity. Here we add clause centrality to these. Small clauses are good because they eliminate many truth assignments and facilitate propagation. Literal Block Distance is defined relative to a CDCL assignment stack, and is the number of different decision levels for variables in the clause. Small LBD clauses are like short clauses relative to assignments that are near the current one [9]. Clause activity is an analog of VSIDS activity, bumped each time the clause is used in a learned clause derivation [103]. Intuitively, clauses with low centrality connect variables "on the edge of the formula", and a long clause with low centrality connects many such variables, so is likely hard to use.

To see the effect of centrality-based deletion on some of the clause quality measures, we measured the quality of learned clauses kept in LOCAL for three deletion schemes: Activity based deletion (default Maple LCM Dist); Centrality-based deletion (HCdel); and LBD-based deletion which was implemented in Maple LCM Dist for this study where it sorts clauses in LOCAL based on LBD and deletes the half with higher LBDs. Table 4.3 shows the results. Reported numbers are the mean of measurements taken just after each clause deletion phase. The last column shows the average solving time of all formulas with each method. Out of 119 formulas that where solved by all 3 solvers, The order of centrality averages holds for all 119 of them. The order of LBD holds for 113 of them and the order of size averages holds for 115 of them.

Deletion method	Clause Centrality	Clause LBD	Clause Size	Time
Activity-based Deletion	106	24	56	401
Centrality-based Deletion	182	15	36	347
LBD-based Deletion	80	9	24	446

Table 4.3: Measures of quality for clauses in the LOCAL clause set, for three deletion schemes. (Centralities are scaled by 10,000).

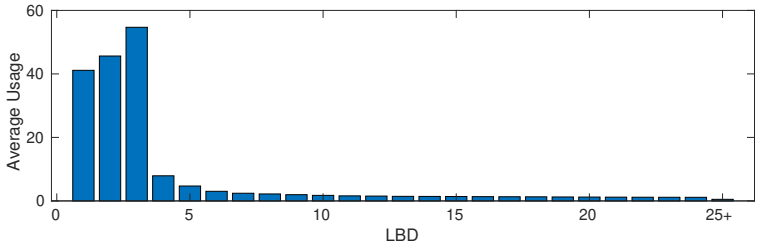
We observe:

- Centrality-based deletion keeps better clauses than activity-based deletion, as measured by both size and LBD, and also performs better.
- LBD-based deletion keeps the "best" clauses measured by LBD and size, has the worst performance and keeps the worst clauses measured by centrality.
- Centrality is the only clause quality measure that perfectly predicts ordering of the deletion methods by solving speed.

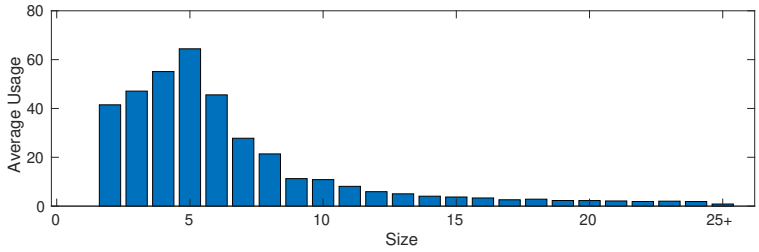
Table 4.3 suggests centrality of saved (not deleted) clauses can be good indicator of the solving time. In the next sections of this chapter, we will look into the usefulness of clauses with high centrality in more details.

4.4 Learned Clause Quality

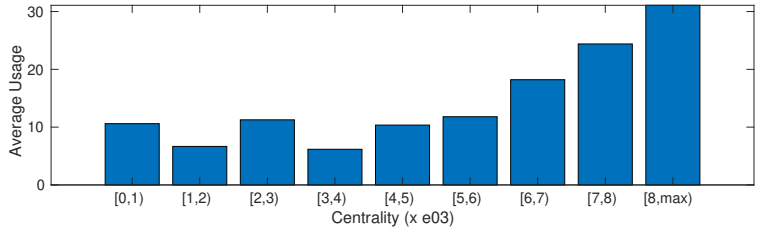
Our experiments in section 4.3 suggests that centrality, can be a useful measure in identifying high quality clauses and using it in deletion. Now we would like take a closer look at the connection between main clause quality measures, Size, LBD and Centrality and their usefulness in conflict analysis. We define Usage to reflect how useful a clause has been in reaching new conflicts and learning new clauses. It is measured by the number of conflicts for which the clause has participated in and appeared in the implication graph in conflict analysis. Size and LBD have been used previously in many clause deletion schemes as a good predictor of usefulness of clauses and we showed, centrality can also be used in clause deletion and result in good performance. Here, we want to compare these measures based on their usage.



(a) Average Usage with respect to clause LBD



(b) Average Usage with respect to clause size



(c) Average Usage with respect to clause centrality

Figure 4.3: Average Usage with respect to different clause quality measures

In the rest of the experiments of this chapter, our base solver for experiments is MapleLCMDistChronoBT which shares the same clause database management scheme of MapleSAT solvers (including Maple LCM Dist) and was the winner in main track of SAT competition 2018 [83, 64]. Consider a modified version of MapleLCMDistChronoBT in which the clause deletion is deactivated and stop this solver after generating 500,000 conflicts. At this point, the modified solvers has learned 500,000 clauses and partitioned and stored them in Core, Tier2 and Local. We ran this solver on 10 randomly chosen formulas from the 2020 SAT competition benchmarks and made a database of 5 million clauses (with a total of about 774 thousands clauses in Core, 695 thousands clauses in Tier2 and 3.526 million clauses in Local). For each clause, we recorded its size, LBD, centrality and usage values. The centrality computations were done using the Brandes [28] algorithm as before but we implemented the algorithm as a preprocessing step in the solver instead of generating the centrality information as an outside process and passing it to the solver as input. This made the computations about 5 times faster and allows computing the exact centrality values for many industrial SAT formulas in a reasonable time. In this experiment, we calculated the exact centrality values in the 10 formulas and have not used sampling.

Figure 4.3 shows the usage rate in clauses with different qualities. Each bin in Figure 4.3 shows the average usage values of clauses with the similar size, LBD and centrality. The first two figures clearly show that as the size and LBD of clauses grows, their usefulness in conflict analysis drops so larger (and higher LBD) clauses are used fewer times in conflict analysis. Figure 4.3c similarly shows that as the centrality of clauses increases, their usage also increases. This clearly suggests more central clauses are more useful in conflict analysis which can explain the good performance of HCdel in Section 4.3.

It is interesting and unexpected to see the huge different in usage between clauses with LBD ≤ 3 (basically clauses in Core) and LBD > 3 in figure 4.3a. We think the sudden change can be explained by the way the LBD values are updated. Every time a clause that does not belong to Core is used in conflict analysis and its LBD value decreases, the value will be updated and the clause moved to Core if it meets the threshold. This means clauses that are used more often, have a higher chance of being moved to Core so more useful clauses with low LBD appear there whereas the clauses that have current low LBD but are not used in conflict analysis, will never get their LBD updated. Our back up experiments show a similar pattern (with low LBD clauses having higher usage value) even if we only consider the initial LBD value of all clauses and never update the values but the change is not as sudden.

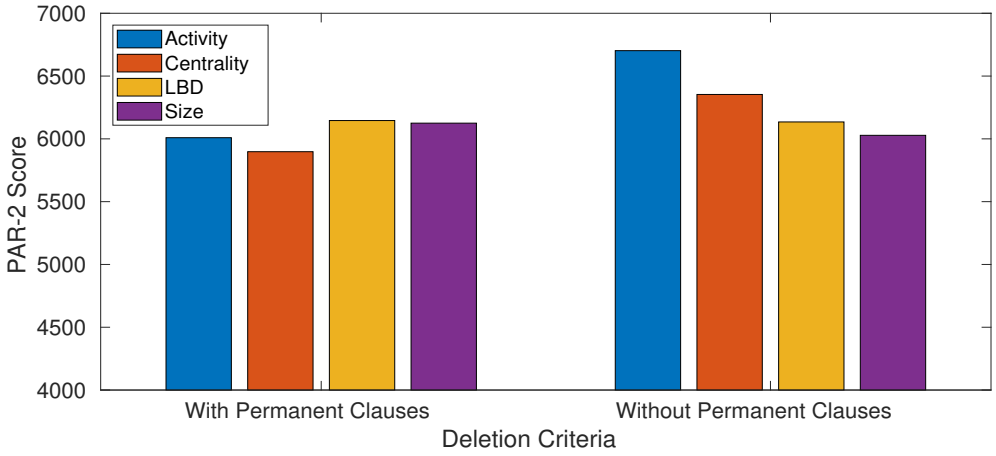


Figure 4.4: Comparison of PAR-2 scores of solvers using different deletion criteria with and without permanent clauses in Core.

4.5 Deletion Criteria in Delete-Half Schemes

In our solver with centrality-based deletion, HCdel, we showed that modifying the Delete-Half scheme to delete the clauses with low centrality (versus clauses with low activity) can be beneficial to performance. It is important to remember that the clause database management heuristic in MapleSAT solvers, keeps clauses with low LBD in Core and Tier2 and the deletion is not purely based on activity. The same holds in HCdel. In this section, we will try to answer the question of which clause quality measure is best if used separately. For that, we continue using the MapleLCMDistChronoBT solver and various modifications of it. We make 8 different modified solvers where each one uses a different criterion in its Delete-Half scheme. 4 solvers keep the Core setting of MapleLCMDistChronoBT and store clauses with $LBD \leq 3$ in Core database permanently. In their deletion, we use the four different criteria of activity, centrality, LBD and size to sort clauses in Local and remove the half with lower qualities. This deletion scheme is repeated in the other 4 solvers with the difference of not having a Core database and storing all clauses in Local. Note that for simplicity, we removed the Tier2 database from all solvers and clauses that are normally stored in Tier2 are stored in Local.

For our experiments in this section, we used the main track of 2020 SAT competition benchmarks. There is a total of 400 SAT formulas and the centrality computation algorithm implemented in the solvers could find exact centrality values of 168 formulas in less that 150 seconds which is our centrality computation timeout. We ran the 8 solvers mentioned above on these 168 formulas with the standard timeout of 5000 seconds and used PAR-2 scores to compare their performance. Note that the solving time of the two solvers using centrality includes the time spent on computing centrality values.

Figure 4.4 compares these solvers using different criteria for deletion based on their PAR-2 scores. The 4 solvers on the right are showing the performance of solvers in Delete-Half scheme when they only use one criteria in deletion. We can see that LBD and Size show the best performance followed by centrality. Sorting and deleting clauses only by activities results in the worst performance among them. The 4 solvers on the left on the other hand are storing low LBD clauses permanently in Core so with Delete-Half scheme on clauses in Local, they are using a combination of two criteria in their clause database management and deletion. The colors indicate different criteria in sorting and deleting of Local. By saving low LBD clauses (which are also usually small) permanently, LBD is no longer the best deletion criterion. In that case, we can see that deletion by Centrality shows the best performance followed by activity and size. This can confirm the good performance of HCdel implemented on another solver (MapleLCMDistChronoBT) and another set of benchmarks. It is interesting that LBD-based deletion in presence of a permanent set of clauses like Core, is even poorer than deleting by LBD and keeping low LBD clauses in Core at the same time.

We can see that in general, solvers in Figure 4.4 that have a permanent set of clauses in Core, perform better that the solvers without Core. In the next chapter, we will focus on permanent clauses and their effect in CDCL solvers.

4.6 Summary

Previously, we demonstrated that betweenness centrality of variables can be utilized in decision heuristics of CDCL solvers to improve their performance. Wondering whether this measure can be used in clause database management heuristics as well or not, led us to introduce a new clause quality measure, clause centrality, and study the result of using it in the clause deletion strategies. To evaluate the effect of utilizing clause centrality in solvers, we introduced a centrality-based modification to standard CDCL deletion heuristic, Delete-Half, and implemented it in Maple LCM Dist, the first-place solver from the Main Track of the 2017 SAT Solver Competition. Our modifications improved the performance on the formulas from this track. We also confirmed this observation using another solver, MapleLCMDistChronoBT which is the winner of main track of SAT competition 2018.

We presented other evidence that clause centrality is an interesting clause quality measure. We showed that the clauses with higher centrality values are used more frequently in conflict analysis which can be one of the reasons of its effectiveness as a measure of quality. We believe that further study of this measure will be productive. We also show that saving low LBD clauses permanently has a positive effect on performance regardless of the criteria used in deletion which we will discuss in details in the next chapter.

Chapter 5

Permanent Clauses

5.1 Overview

Modern CDCL solvers learn a new clause after each conflict. They normally learn and add thousands of clauses per second [87, 89]. These clauses are stored in learned clause database and keeping all learned clauses can be impractical as they will have huge memory consumption which may not be available since the growth of learned clauses can be exponential in the number of variables of the formula [24]. Other than limited memory problem, it is costly to perform BCP on clauses and not all clauses are valuable enough to be worth the cost. The clause database management heuristic deletes clauses periodically to help keep the solvers efficient [9, 6]. It is important to have good "clause quality" measures that can predict future value of clauses and use them in clause deletion techniques. Most solvers store some "high quality" clauses permanently and review the others periodically to delete some of lower quality. We will call the set of clauses that are never deleted PERM for Permanent clauses, and the set which is reviewed for deletion TEMP for Temporary clauses. Technically, some PERM clauses can be deleted because they are satisfied by a learned unit clause (so they are satisfied at decision level 0), but these clauses could not be used again anyway even if they were kept. PERM and TEMP are often, but not always, stored in distinct data structures.

In the previous chapter, we introduced a clause quality measure based on betweenness centrality of variables and used it in deletion scheme from TEMP. In this chapter, we show how we can utilize this measure in PERM, but first we study the PERM set and its properties in current state of the are solvers in more depth. Here, we focus on PERM clauses learned by solvers of the MapleSAT family [71, 90, 69, 64, 83, 114] , which have performed very well in recent SAT solver competitions. Our experimental results convinced us that the clause deletion scheme is important to their performance, but the complexity of the clause database management heuristic makes it hard to understand why. Recent MapleSAT-based solvers have three distinct stores of clauses, use at least two dynamic clause quality measures (LBD and activity), and a number of heuristic rules to move clauses between the stores. In

contrast, it is desired to build quite good solvers with much simpler schemes [58, 57] which will be discussed in the next chapter of this thesis.

Many solvers place only binary clauses in PERM and use a simple measure to delete from TEMP. For example, Glucose and Cadical use LBD with ties broken by size [9, 22]. Cadical also retains some TEMP clauses based on two bits of recent use information. There are other solvers that allow more flexibility in their PERM criteria. For example, MapleLCMDistChronoBT-DL (Duplicate Learnts) [64] which is the winning solver of SAT Race 2019 [52], mark the clauses that are learned repeatedly after deletion as PERM.

We will look at different clause quality measures and define PERM with respect to them. The main quality measures that will be studied and discussed here are Size, LBD and finally Centrality. Size is the number of literals in the clause. LBD [9] is the number of decision levels of literals in the clause at the time it is computed (at which time all literals must be assigned). Clause Centrality [56] as introduced in the previous chapter is the average betweenness centrality of its variables in the primal graph of the formula [55]. In this chapter, we will report an empirical study of PERM in MapleLCMDistChronoBT. We show that:

- Usually PERM is of moderate size, but sometimes it grows very large.
- At least some ways of restricting PERM size for formulas where it gets large did not help.
- Alternate LBD and size based criteria for PERM can improve performance (with similar-sized PERM). In particular (perhaps surprisingly) sending all clauses of size up to 8 to PERM was very effective.
- Adding very high-centrality (HC) clauses to PERM improved performance on formulas for which centrality computation is fast. Our results indicate that very high-centrality clauses are valuable even if they are long.
- The best improvement in our experiments comes from a combination of size≤ 8 and adding HC clauses to PERM. This version solved 197 instances, 13 more than the 184 that MapleLCMDistChronoBT solved.
- There are small clauses that are easy to derive, which help when added to TEMP, but hurt when added to PERM.

The base solver in all reported experiments is MapleLCMDistChronoBT, the first-place solver in the 2018 SAT solver competition [83, 53]. We denote simply "maple" in keys of some figures, to keep names short. The MapleLCMDistChronoBT deletion scheme was originally adopted from COMiniSatPS [64, 90]. Our data are for 400 formulas from the Main Track of the 2020 SAT competition with a 5000 second timeout [12]. Computations were

performed on the Cedar compute cluster [31] operated by Compute Canada [33] which is a cloud service. The cluster consists of 32-core, 64 GB nodes with Intel "Broadwell" CPUs running at 2.1Ghz.

5.2 PERM Set in State of the Art Solvers

We categorize clauses learned by the solver into two categories based on the possibility of them being deleted. We call the clauses that are kept by the solver permanently, PERM and the clauses that are considered for deletion TEMP. If a clauses is marked as PERM it will never change to TEMP but technically there are a small number PERM clauses that are deleted if they are satisfied by a learned unit clause (at level 0) and can never be used again. In this work, we only consider learned clauses even though the initial set of clauses in the formula are also never deleted.

PERM clauses in state of the art solvers are usually defined based on their Size or LBD. In deletion schemes implemented in early CDCL solvers, size was usually the main factor in keeping clauses permanently. For example in Seige [94], the permanent set was binary and ternary (size 3) clauses. In GRASP [78], the deletion was performed on clauses of size larger than 20 so smaller clauses would have been saved permanently. This made a large portion of learned clauses. Over the years the deletion has become more aggressive and there are empirical studies showing keeping the size of the clause database small helps improve performance [9]. MiniSAT [39] started with deleting more clauses at the time and only kept binary clauses permanently in the clause database. Binary clauses are known to be valuable and are still saved permanently by almost all CDCL solvers which makes them the most common definition of PERM clauses and the deletion happens periodically from the rest of the clauses based on their last known LBD or Activity [8, 20, 69]. Kissat, the winner of SAT competition 2020 is also among the solvers that follow MiniSAT's strategy in their permanent clauses [22]. The following properties make binary clause good targets for PERM:

- They don't require much memory to be stored.
- The don't take much BCP time as they will be visited at most once.
- They can easily become unit after one of their literals is assigned to false hence they have a higher chance of triggering unit propagation.

These are interesting features but there are other clauses that can have the same properties to some extent and the question is whether there are more learned clauses worth storing permanently. For example, it is reasonable to ask this question about clauses of size 3 or low-LBD clauses.

In the past few years a new criteria based on LBD for permanent clauses and generally managing clause database has been used by Maple-SAT family solvers (with small variations) which have been the winners of SAT competitions between 2016 and 2019 [1]. MapleSAT adopted the deletion scheme first introduced in a solver called COMiniSatPS [90, 89]. It partitions the clause database into three different sets called Core, Tier2 and Local. The decision of where to store a newly learned clause is based on its LBD at the time; A clause is stored in Core if its LBD ≤ 3 (which can be changed to LBD ≤ 5 if size of Core is too small after the first 100,000 conflict), in Tier2 if $4 \leq$ LBD ≤ 6 and in Local if $6 <$ LBD. A clause may be moved from one set to another based on LBD or usage. The LBD of each clause is recomputed whenever it is used in conflict analysis. If the LBD of a clause at that time is sufficiently reduced and meets the thresholds, it will be moved from Local to Tier2 or Core, or from Tier2 to Core. Based on its updated LBD, every 10,000 conflicts, all clauses in Tier2 that have not been used during the last 30,000 conflicts are moved to Local. The deletion only happens to clauses stored in Local using the Delete-Half scheme with the Activity as clause quality measure for sorting and deletion. The clauses stored in Core make the PERM set and will be saved permanently. In the rest of this chapter, our experiments will be on the MapleLCMDistChronoBT solver using this scheme as clause database management heuristic.

5.3 Usage in Learned Clauses

Now that we explained the partition scheme in MapleSAT solvers, we want to see how are these sets different. The obvious difference is their LBD values but to compare them in other ways we designed the following experiment. In all the experiments of this section, our base solver is MapleLCMDistChronoBT which shares the same clause database management scheme of MapleSAT and was the winner of SAT competiton 2018 [83, 64].

Consider a modified version of MapleLCMDistChronoBT solver in which the clause deletion is deactivated and stop this solver after learning 500,000 conflicts. At this point, the modified solvers has learned 500,000 clauses and partitioned and stored them in Core, Tier2 and Local. We ran this solver on 10 randomly chosen formulas from the 2020 SAT competition benchmark and compared the average value of different quality measures in the clauses of these sets with each other. Note that we are comparing 5 million clauses here generated from 10 SAT instances. Out of them, there is a total of 774 thousand PERM clauses in Core, 695 thousand TEMP clauses in Tier2 and 3.526 million TEMP clauses in Local. Table 5.1 compares the average value of LBD, Size, Age (number of conflicts since generation), Centrality and different usage measures in them. We considered 3 different usage measures, UP Usage, CA Usage and Activity defined as follows:

DataBase	LBD	Size	Age	UP Usage	CA Usage	Activity	Centrality
Core(PERM)	2.27	6.47	267215	199	49	1.52	1.59
Tier2(TEMP)	5.10	11.26	229957	13	5	1.40	1.68
Local(TEMP)	16.50	37.11	253054	1.8	1.5	0.24	1.79

Table 5.1: Average value of clause quality measures in solver with no clause deletion

UP Usage: UP (Unit Propagation) Usage is used to reflect how useful a clause has been so far in activating unit propagation. This happens when a clause becomes unit and has just one unassigned literal under some variable assignment of the solver. The UP usage is basically counting these occurrences for each clause. UP usage is not really a quality measure used by deletion methods but it is safe to assume smaller clauses have higher chance of becoming unit under random assignments.

CA Usage: CA (Conflict Analysis) Usage is mainly used to reflect how useful a clause has been in reaching new conflicts and learning new clauses. The CA Usage is measured by the number of conflicts the clause has participated in and appeared in the implication graph.

Activity: Activity here is defined as the VSIDS Activity [81]. It is a similar measure to CA Usage that takes into account the time of usage as well. The Activity values are reduced periodically. Hence, the clauses that were used more recently have higher activities and are considered to be more useful.

Table 5.1 shows the average value of different clause quality measures in the three learned clause databases. Activity and Centrality values are normalized by 10^3. Given that PERM clauses were originally stored in Core based on their LBD values, it is expected that they have lower average LBD values and given that LBD is bounded by Size, it is easy to see why they are smaller as well. Table 5.1 suggests clauses in Tier2 have the lowest average age whereas clauses in Core appear to have the higher average age. We know the inactive clauses in conflict analysis in Tier2 are moved to Local by the solver so clauses left in Tier2 (shown in the table) are the more active ones. On the other hand, it has been shown empirically that the CA usage of clauses drops quickly over time [58] so it can also explain why the average age of clauses left in Tier2 is lower than other stores. It is not clear why the average age in the clauses of Core (PERM) is higher than the rest of the clauses, especially when we look at the average age of PERM clauses (which is 267215) compared to the average age of TEMP clauses (which is 249251). In an attempt to explain this difference, figure 5.1 shows a histogram of the average LBD of clauses with respect to their age. It shows the LBD of learned clauses increases over time which means LBD of clauses learned around the beginning of the run is lower hence more PERM clauses are generated then.

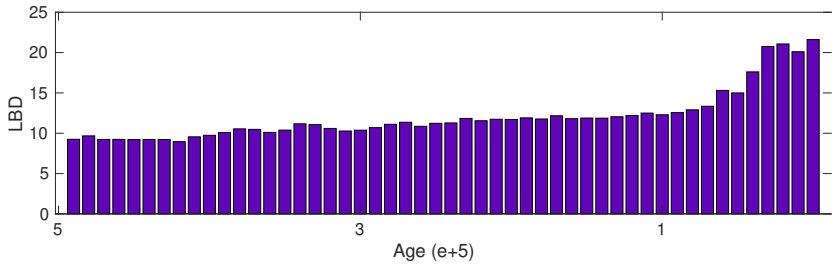


Figure 5.1: Average LBD of learned clauses with respect to Age

Going back to table 5.1, it is interesting to see that PERM clauses learned by MapleLCMDistChronoBT are used much more than TEMP clauses in both unit propagation and conflict analysis. Since they have smaller size (and LBD), we know that they have a higher chance of becoming unit (a clause of size $|n|$ becomes unit after $|n| - 1$ of its literals are assigned) and it is reasonable to see higher UP Usage in PERM clauses. It is interesting to see that PERM clauses are also used much more than TEMP clauses in conflict analysis and are more useful in that sense as well. This can explain why low LBD clauses are very important to solvers.

Activity which is computed based on CA usage (and age) is also higher in PERM clauses. MapleLCMDistChronoBT performs deletion from the clauses in Local based on activity values and if we turn on the deletion step in this experiment and calculate the average activity values again, we see that the average activity values of TEMP clauses in Local increases from 0.24(e-15) to 77.44(e-15) even though the difference in activity of PERM clauses remains almost the same. This is because most of the clauses in Local with smaller activity values are deleted and only the most active ones remain. Finally, looking at the average centrality values of PERM and TEMP clauses, we can not see any meaningful difference in them but TEMP clauses seem to have slightly higher centralities.

5.4 Size and Value of PERM in MapleLCMDistChronoBT

After studying some measures of usefulness of PERM clauses in MapleLCMDistChronoBT, we want to look more into individual SAT instances to see how PERM clauses look for them. It is known that the industrial SAT instances have different characteristics and they usually hold some sort of structure as opposed to random instances [72]. We would like to study these formulas in terms of the size or value of PERM in MapleLCMDistChronoBT. To do that, first we looked into the size of PERM set in different instances from 2020 SAT competition benchmarks. MapleLCMDistChronoBT solved 184 instances from this benchmark set with the 5000 second time-out. Figure 5.2 is a histogram showing the number of PERM clauses that are stored in Core at the end of the run on these formulas.

For about half of the formulas the final size of PERM is 30,000 or less, which is moderate compared to the average TEMP size of about 30,000. However for nearly a quarter of the

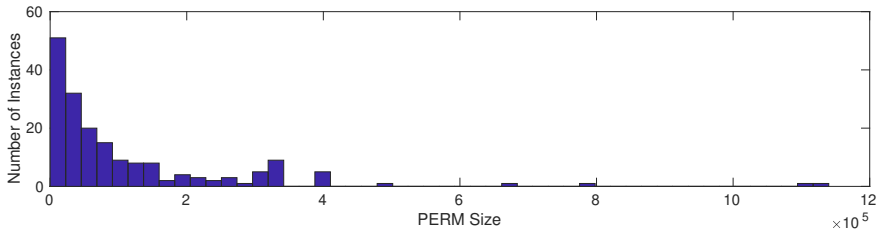


Figure 5.2: PERM Size Histogram

Benchmark Set	PERM Size	Conflict learning rate	Decision making rate	GLR
Small PERM(SP)	11801	7661	111281	0.33
Large PERM(LP)	349364	3317	20965	0.41

Table 5.2: Learning rates in instances with Large/Small number of PERM clauses

instances (22%), the final PERM size is more than 150,000. We call these, LP (Large PERM) instances. In LP instances, the PERM clauses make a large fraction of all learned clauses stored by the solver. Let's also define SP (Small PERM) instances to be the instances that have less than 20,000 PERM clauses by the end of the solver's run. There is a total of 59 SC instances in this set (32 %).

To see the effect of a large PERM set on solver progress, we calculated both average conflict learning rate (#conflict/time in seconds) and decision making rate (#decisions/rime in seconds) in these sets and the results, as shown in table 5.2, confirm that both rates are smaller when the PERM size is larger so the learning has been slowed down in this sense. It is reasonable to expect that the solving time of LP instances is larger that of SP instances which is why they have more learned clauses. It is actually true but we would like to point out that the average solving time of the LP instances is almost 4 times larger than the average solving time in SP instances whereas the number of conflicts occurring in this time is only about twice as much. This is clear by comparing the conflict learning rates as well.

Another way to compare the performance in these two sets is by measuring the Global Learning Rate (GLR) of the solver for these instances. GLR was introduced in [72] as a performance measure for solvers and is computed as follows: $GLR = \#Conflicts/\#Decisions$. Larger GLR in a solver means it made fewer decisions to learn a new clause (reach a conflict) which is considered a positive learning factor. The average GLR is 0.33 in SP instances and 0.41 in LP instances which is slightly higher and somewhat indicating having more PERM clauses is better in terms of GLR. This is not surprising as having a larger clause database results in more BCP with each decision and so solvers can reach a conflict with making fewer decisions.

We report two experiments to evaluate the usefulness of PERM clauses in MapleLCMDistChronoBT. In the first experiment, we compared the default solver with two modified versions, one with PERM empty, and one with only binary clauses sent to PERM. We ran these solvers on the full benchmark set with 400 instances to see the effect of reducing the size of PERM on performance. Figure 5.3 shows that both modifications reduce performance substantially illustrating the value of PERM clauses in MapleLCMDistChronoBT.

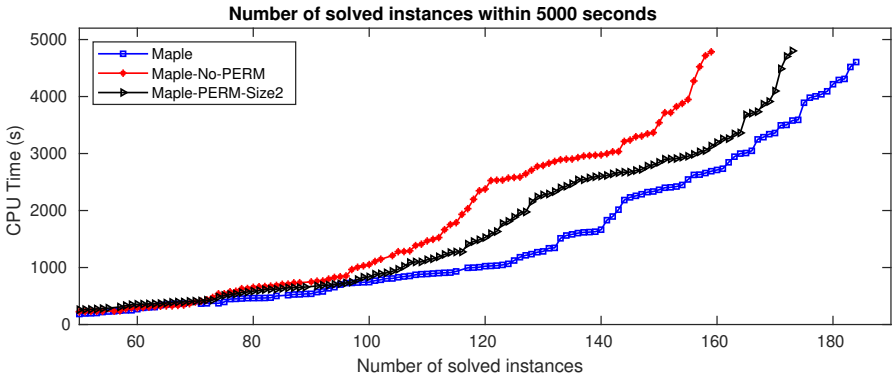


Figure 5.3: Effect of removing PERM from MapleLCMDistChronoBT

Conventional wisdom suggests 150,000 is very large for a learned clause set, and we hypothesized it might be better to limit its size. In the second experiment, we use various schemes to restrict the size of PERM, with the goal of keeping it less than 100,000. We applied these schemes to the formulas for which PERM grew to more than 150,000 clauses (LP instances). As Figure 5.4 shows, even for these formulas most of the methods were damaging to performance, and even the best do not seem beneficial. The schemes are as follows:

- Maple-PERM-LBD2: This modification changes the criteria for PERM clauses and only stores clauses with $LBD \leq 2$ in Core permanently .
- Maple-PERMset-max100K: If size of PERM reaches 100,000, the solver sends no more clauses to PERM and sends all new clauses to TEMP.

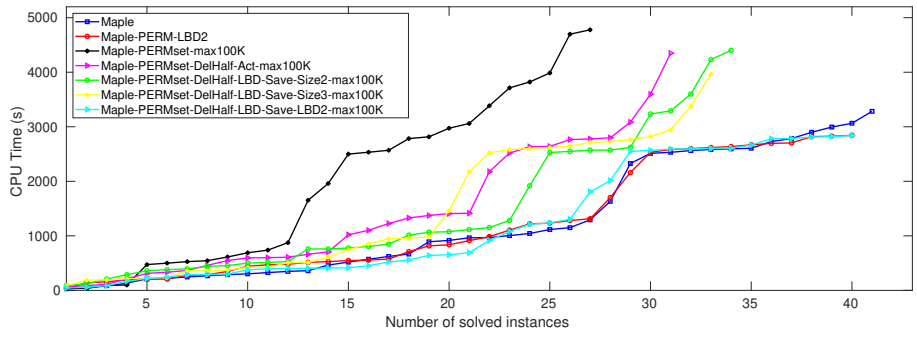


Figure 5.4: Effect of limiting size of PERM on "LC" formulas.

- Maple-PERMset-DelHalf-Act-max100K: If size of PERM reaches 100,000, the solver invokes a Delete-Half deletion scheme on PERM, based on clause activity.

- Maple-PERMset-DelHalf-LBD-Save-X-max100K: If size of PERM reaches 100,000, the solver invokes a Delete-Half deletion scheme on PERM, based on LBD (with ties broken by clause age), but never deleting clauses with property X, for X in $\{\text{size} \leq 2, \text{size} \leq 3, \text{LBD} \leq 2\}$.

We also show the average PERM size in these solvers in figure 5.5 to illustrate the reduction in size. The rate of learning PERM clauses over all learned clauses in MapleLCMDistChronoBT and the modified solvers is about the same (around 14 %). These preliminary experiments suggest that keeping all PERM clauses might be useful, even if they make the size of clause database very large.

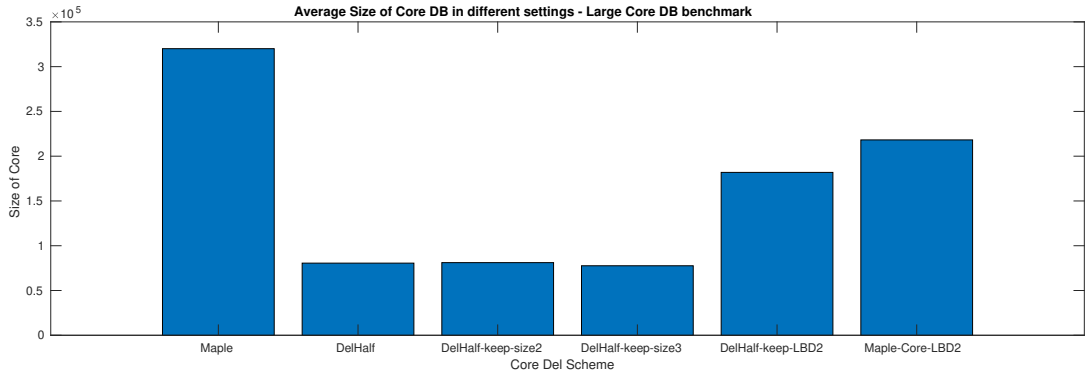


Figure 5.5: Average size of the PERM set at the end of solver's run on LC benchmarks

5.5 Varying Size and LBD Criteria for PERM

The MapleLCMDistChronoBT criterion for putting in PERM is LBD ≤ 3. (Sometimes it is changed to LBD ≤ 5 during the run but for few formulas.) We report an experiment in which we vary the criteria over two ranges: Size $\leq k$, for $k \in \{2, \ldots 15\}$ and LBD $\leq k$, for $k \in \{2, \ldots 8\}$. First, we would like to show how the change in PERM criterion effects the average size of PERM. Figure 5.6 shows the effect on the fraction of learned clauses sent to PERM, and on the final size of PERM where yellow bars indicate these values in solvers in which PERM is defined based on Size and purple bars show the solvers using LBD as PERM criteria. The purple bar with L3 label shows the closest solver to MapleLCMDistChronoBT's clause management scheme which ends up with a PERM size of 70,000 on average.

Figure 5.7 shows the effect of these variations on PAR-2 performance scores. The PAR-2 score is the main criteria used in SAT competitions to evaluate the performance of different solvers [1]. It is computed as the average solving time of a solver where the instances that could not be solved within the time cut off t (5000 seconds here), are given a time of

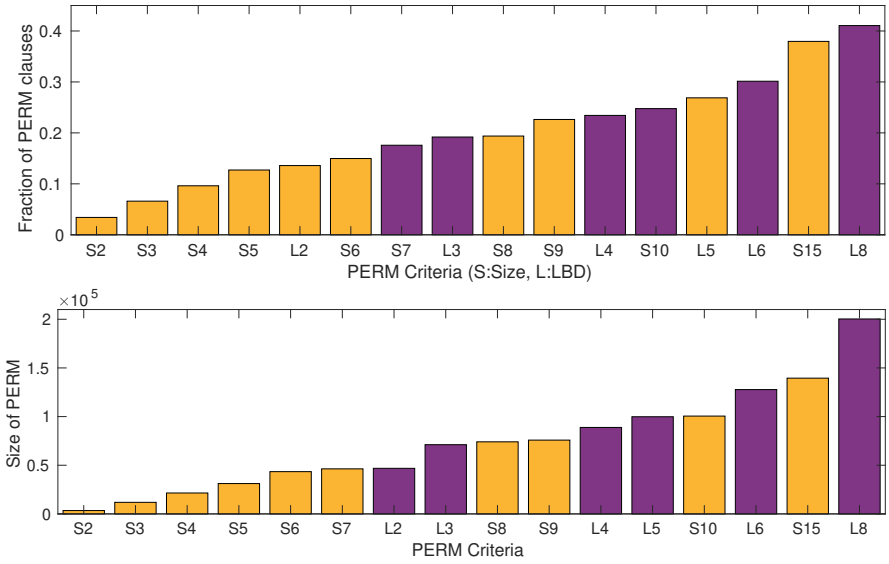


Figure 5.6: Fraction of learned clauses sent to PERM (upper), and final size of PERM (lower) with varied PERM criteria.

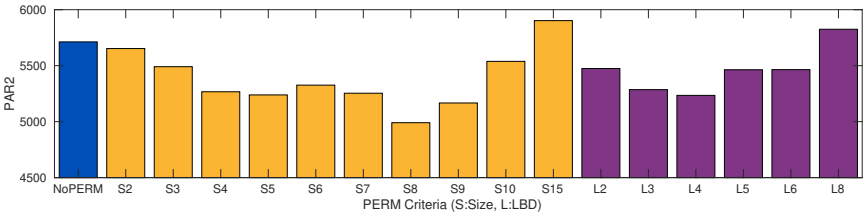


Figure 5.7: Effect of PERM criterion on PAR-2 Scores

$2t$. The lower the value of PAR-2, the faster and more efficient the solver has performed. The first bar (blue) in figure 5.7 shows the PAR-2 score of a solver with no PERM that considers all clauses as TEMP, stores them in either Tier2 or Local and considers them for deletion periodically from Local. We have not changed the way the solver manages these two databases other than changing the lower threshold of Tier2. Figure 5.7 suggests that having PERM clauses helps with performance but as the PERM set gets too large, the performance starts to drop again. The best PAR-2 score belongs to the solver with Size ≤ 8 as PERM criteria. As Figure 5.6 shows, the number of PERM clauses in solver with Size ≤ 8 falls between that for LBD ≤ 3 and LBD ≤ 4. Figure 5.8 compares the performance of this solver with MapleLCMDistChronoBT in a cactus plot.

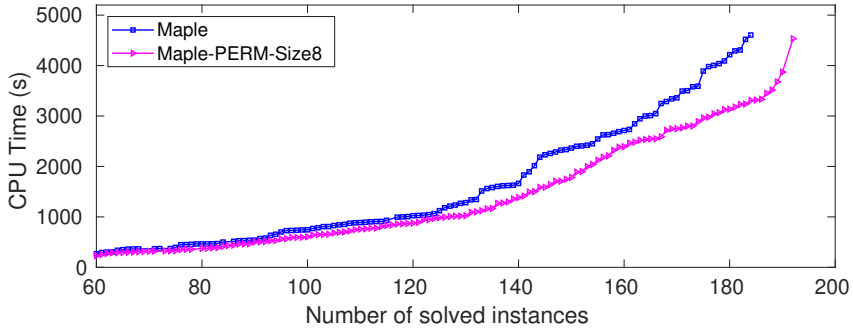


Figure 5.8: Performance of solvers with small clauses in Core

5.6 Adding High-Centrality Clauses to PERM

Clause betweenness centrality as defined in Chapter 4 has shown to be a useful clause quality measure. We showed that clauses with high centrality (HC) values were used more often in conflict analysis, suggesting they are more useful in generating new conflicts and learning clauses. We also showed they can be utilized in clause deletion schemes to improve performance so here, we want to see if they can help if used in PERM clauses as well.

Here we add a limited number of high-centrality (HC) clauses to PERM in MapleLCMDistChronoBT. We computed variable centralities using the Brandes algorithm [28] in the solver as before. The centrality computation sometimes takes too long, and we limited it to 150 seconds, obtaining centralities for 168 of the 400 instances. For the other formulas we did not use centrality. We normalize the centrality values by $1/(n-1)(n-2)$ where n is the number of variables, so they fall in $[0, 1]$. The HC clauses can be large and result in computation and memory overhead so care is needed when adding them to PERM. We aimed to include at least the 0.02% of learned clauses with highest centrality. We set an initial centrality threshold of $CT \geq 0.008$ which was chosen empirically. Every 100,000 conflicts, if the number HC clauses in PERM is less than 0.02% of all learned clauses, CT is reduced by 0.001, but it is never reduced below 0.001. We report three versions:

- Maple-PERM-HC-max10K: Add at most the first 10K HC clauses to PERM.
- Maple-PERM-HC-max25K: Add at most the first 25K HC clauses to PERM.
- Maple-PERM-HC-Size15-max10K: Add HC clauses to PERM only if they have size ≤ 15, adding at most the first 10K.

Figure 5.9 compares the performance of these versions, including centrality computations, against the default solver. The two versions with no size limit on the HC clauses performed noticeably better. The version with the size limit performed only slightly better than the default. This indicates long HC clauses are valuable. The average number of additional HC PERM clauses (that would not have been placed in PERM because of LBD) was 8,200 in the version with the limit of 10K, 16,000 with the limit of 25K, and 7050 with the limit of 10K and size ≤ 15.

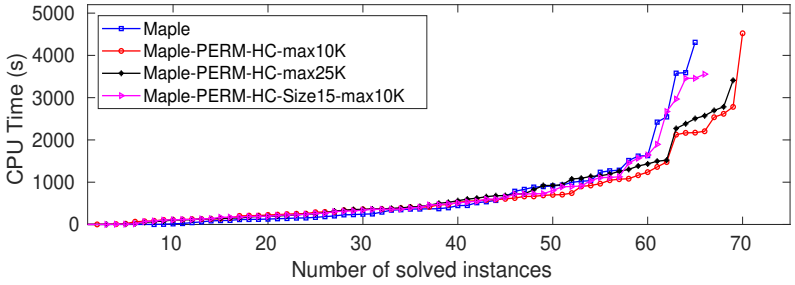


Figure 5.9: Performance of solvers with high-centrality clauses in PERM

Long clauses are generally less valuable than short clauses. They eliminate fewer truth assignments, and are used less frequently by CDCL solvers. However, it is interesting to see that our data shows that the value of HC clauses added to PERM comes from the longer ones as the modified solvers that stores only smaller HC clauses in PERM does not perform as well. So far we have shown changing the PERM criteria from $LBD \leq 3$ to $Size \leq 8$ and also adding HC clauses to PERM can help performance of the solver. Often the combination of two heuristics that are beneficial does not improve over the best of the two. However, in the case of our criteria for PERM, the following combination did improve overall performance: For each instance, if we did not get centralities within the 150 seconds time limit, we used PERM criterion of Size$\leq$ 8; otherwise we used LBD$\leq$ 3 and added HC clauses. Figure 5.10 compares this version with the original and shows it improves the performance by solving 13 more instances.

5.7 Small good clauses not to add to PERM

In Section 5.4, we showed that performance improved when we saved all learned clauses of size up to 8 to PERM. Here, we show that there are small clauses we can derive simply

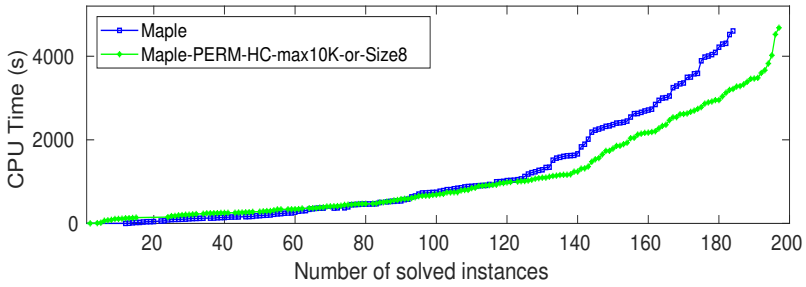


Figure 5.10: Performance with PERM criteria Size$\leq$ 8 or HC.

which help when added to TEMP but not when added to PERM. Standard conflict analysis schemes derive one clause, called the 1-UIP (for First Unique Implication Point) clause, at each conflict. Various other schemes have been tried, but most reports confirm that the 1-UIP scheme is best [113, 38, 102]. An example of adding more clauses is in [38], but these clauses require significant additional reasoning.

Here, we introduce a simple scheme to learn additional small clauses which are very cheap to obtain but still improve performance. Regarding the focus of this paper, the interesting observation is that they have length less than 8, but adding them to PERM reduces performance while adding them to TEMP improves performance. (In contrast, adding all small 1-UIP clauses to PERM improves performance, as we showed above.) Assume a conflict at level x, meaning after assigning x literals $l_1, l_2, .., l_x$ to true, a conflict is reached. After conflict analysis the solver backjumps to a level b and learns a 1-UIP clause $C_1 = \{m_1, m_2, ..., m_{i-1}, m_i\}$. Only one literal m_i from C belongs to level x, and $b < x$, so after the first b decisions, if we had C_1 in the clause database, unit propagation could prevent this conflict by assigning m_i to true. Therefore, we can also learn clause $C_2 = \{\neg l_1, \neg l_2, ...\neg l_b, m_i\}$. If $b < 6$, this clause has size $<= 6$, so we have a new small clause with little work. Here the last two literals of C_2, $\neg l_b$ and m_i, are glued together so C_2 has LBD $|b|$ and size $|b| + 1$. We modified the solver to learn clauses of this kind and added them either to PERM or TEMP.

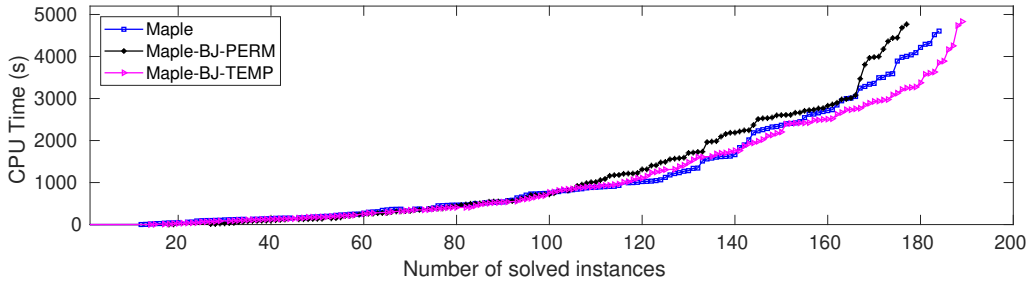


Figure 5.11: Performance of solvers with new learned clauses

Figure 5.11 compares performance of these versions with MapleLCMDistChronoBT. In the formulas of this experiment, an average of 8500 additional clauses were generated using this scheme most of which had LBD 4 or 5. This can be a factor in making them less interesting for PERM.

5.8 SAT vs. UNSAT formulas

In the final section, we briefly show the effect of different modified solvers discussed in this chapter on satisfiable and unsatisfiable formulas. Table 5.3 shows the total number of formulas solved by the main interesting solvers introduced in this chapter distinguishing between satisfiable (SAT) and unsatisfiable (UNSAT) formulas. The first line in the table shows that MapleLCMDistChronoBT solved a total of 184 formulas from the 2020 competition benchmarks of which 86 were satisfiable and 98 were unsatisfiable. Just by changing the PERM criteria from $LBD \leq 3$ to $Size \leq 8$ we improved the number solved by 10 formulas where most of the improvement was from satisfiable instances.

Our solver with centrality based modifications in PERM, Maple-PERM-HC-max10K, also shows a similar pattern with 4 more satisfiable instances and just 1 more unsatisfiable instance. Note that for this solver, if the centrality computation was not completed within the time limit, the solver continues as MapleLCMDistChronoBT without any modifications.

The difference is less strong in solvers with the additional learning scheme. Learning additional clauses with this scheme when added to TEMP, improved performance and it solved 5 more formulas than MapleLCMDistChronoBT from which 3 are satisfiable instances and 2 are unsatisfiable.

Solver	# Solved	SAT	UNSAT
MapleLCMDistChronoBT	184	86	98
Maple-PERM-Size8	**194**	**94**	**100**
Maple-PERM-HC-max10K	189	90	99
Maple-BJ-PERM	177	81	96
Maple-BJ-TEMP	189	89	**100**

Table 5.3: Performance on Satisfiable *vs* Unsatisfiable instances.

5.9 Summary

In this chapter, we started by comparing PERM and TEMP in MapleLCMDistChronoBT with respect to different quality measures. We showed that PERM clauses are on average smaller with lower LBD values and are used substantially more in conflict analysis and triggering unit propagation. However, they do not show a meaningful difference with respect to centrality values.

We showed that the size of PERM in some industrial formulas grows quickly resulting sometimes in more than million clauses stored permanently. Wondering whether this helps or hurts the performance of solvers, we tried various modification of MapleLCMDistChronoBT to limit the size of PERM but showed that all of them damaged the performance. We also tried comparing the PERM definition of MapleLCMDistChronoBT with a solver with no PERM and a solver with binary clauses as PERM (similar to most solvers) and showed that MapleLCMDistChronoBT performs substantially better than both modified versions.

We compared different criterion for PERM based on size and LBD and showed $LBD \leq 4$ and $Size \leq 8$ result in best solvers. Based on our findings on the usefulness of High Centrality (HC) clauses in conflict analysis and also the effect of using them as a quality measure for deletion, we proposed a solver where HC clauses are stored as PERM and showed it will result in performance improvements on industrial formulas. Interestingly when we only used the small HC clauses, the improvements were not as strong suggesting long HC clauses had meaningful value. We also showed that there are additional small clauses other than the 1-UIP scheme that can be learned efficiently by the solver with low cost. These can be beneficial to solver if added to TEMP but not to PERM.

Finally we showed most of the improvements made by these solvers are in satisfiable instance with less effect on unsatisfiable ones.

Chapter 6

Simplifying Clause Database Management

6.1 Overview

CDCL SAT solvers generate a very large number of new "learned" clauses, so clause management methods are important to solver performance [6, 87]. In particular, most learned clauses must be deleted to keep the clause database of practical size, and the clause database management scheme is one of a small number of key heuristic mechanisms in a CDCL solver [89, 9]. Typical clause maintenance strategies involve two sets of learned clauses, which we call PERM and TEMP as discussed in chapter 5. In the MapleSAT solver family, TEMP clauses are stored in Local or Tier2 and PERM clauses are stored in Core (clauses placed in Core are retained for the entire run). The size of Core is limited by being selective about which clauses are added. The large majority of learned clauses are placed in Local. The size of Local is limited by periodic deletion of "low quality" clauses, which are deemed unlikely to be of high future utility. The quality measure is typically a combination of size, age, literal block distance (LBD), centrality and some measure of usage or activity [9, 39, 46, 90, 56, 89, 69].

In our work on clause database management schemes that was discussed in previous chapters, we realized that major changes to the general scheme are rare, but over time many refinements have been combined to make the overall mechanism in the best recent solvers quite complex. Most details have intuitive explanations, and were chosen based on empirical performance results. At the same time, the complexity seems perhaps a bit much relative to our understanding of "clause quality". This complexity makes it hard to evaluate the contributions of individual elements of clause management, and is an obstacle to adding new features or refined quality measures. In this chapter, we address this problem by introducing a new, simple, clause management scheme.

There are two main aspects to a clause deletion strategy. The first is a method to categorize clauses as likely to be useful (high quality), or not (low quality). The second

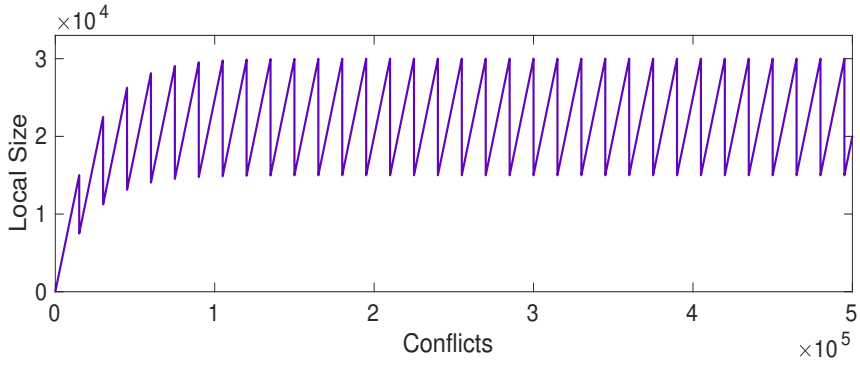


Figure 6.1: Number of TEMP Clauses with Delete-Half Reduction Scheme.

is implementation of an algorithmic method to remove low quality clauses efficiently. In an idealized scheme, we might have a clause quality measure Q, and keep the clauses in a heap so that the lowest quality clause(s) can be removed when the clause database is deemed too large. Conventional wisdom is that using a heap would be too inefficient. It also seems unlikely that spending time to obtain the very worst clause is necessary. Thus, fast heuristics are desired. One scheme, which we call Delete-Half, is to periodically sort the clauses of Local and delete the half with lowest quality. This scheme has been very widely used for many years, but there are many other possible schemes. While some solvers use other schemes (e.g., [19, 101]), we think much more investigation is justified. Regarding clause quality, we expect a very good clause quality measure to involve a combination of many factors. The dominant current quality measure uses VSIDS-like clause activities or LBD. Unfortunately, the way activities are computed and maintained in practice makes it hard to combine activity with other measures of quality in a simple and meaningful way.

Before proceeding, we make some observations about the Delete-Half scheme. First, the size of the resulting Local clause set has a "saw-tooth" pattern, as indicated in figure 6.1. This figure shows the number of TEMP clauses in MapleLCMDistChronoBT solver [83] but other solvers with Delete-Half scheme show the same pattern. While it is possible that this pattern is somehow beneficial, in the absence of evidence to suggest so, we should assume that a uniform "ideal" clause set size would be as good or better. Second, in recent solvers, the clauses are sorted for deletion every few thousands conflicts and the store has a few tens of thousands clauses at each deletion. Assuming a sorting cost of $nlogn$, this could be expensive in terms of computation time. Third, the current scheme, as it has been refined for various goals, has actually become rather complex, with much of the complexity perhaps not well justified. This can be partly due to the fact that usually solvers are build upon older versions with various modifications and details can be missing and very hard to track and reevaluate every time. Forth, a new clause may be deleted very soon after being added, if it is not used right away, so newly learned clauses may not be kept long enough for the

algorithm to discover that they are useful. This is especially a problem with solvers that use LBD for sorting in the Delete-Half scheme.

Our goal in this chapter is to identify simple methods that might largely account for effectiveness of the best current schemes. We show:

- We can replace the popular Delete-Half clause deletion scheme by a much simpler scheme which we call "online" clause deletion. It is simple to implement and maintains the size of Local at any desired value. It does not use sorting and in many natural instantiations takes constant time per conflict. The scheme is presented in Section 6.2.

- A simple instantiation of this scheme performs comparably to the state of the art solvers. In particular, we implemented the scheme within MapleLCMDistChronoBT, the first-place solver from the 2018 SAT Competition [83, 1], and obtained performance almost the same as the original, despite very little effort at optimization or "tuning".

- This instantiation takes into account clause usage and LBD using very simple mechanisms. The resulting solver (Online-RU-T2Flag) and its performance are described in Section 6.5.

- The data from a number of experiments measuring performance or other properties, to aid in understanding the degree to which the particular methods play a role in solver performance. These appear throughout remaining sections.

6.1.1 Performance Evaluation and Base Solver

Our performance evaluations are carried out using the 400 formulas from the main track of the 2018 SAT Solver Competition, with a 5000 second cut off. The computations were performed on the Cedar compute cluster [31] on 32-core, 128 GB nodes with Intel "Broadwell" CPUs running at 2.1Ghz. Similar to chapter 5, our baseline solver for performance evaluation is MapleLCMDistChronoBT, winner of the SAT 2018 competition and all other solvers in our experiments are modified versions of it.

The clause deletion scheme of MapleSAT family [69, 83, 89, 90] has been discussed in detail in previous chapters but here is a general description of it as a reminder. The scheme has three clause databases, called Core, Tier2 and Local. Core stores PERM clauses and TEMP clauses go to Local or Tier2. The decision of where to store a newly learned clause in is based on its LBD: Core if LBD ≤ 3, Tier2 if $4 \leq$ LBD ≤ 6 and Local if $6 <$ LBD. A clause may be moved from one DB to another based on its updated LBD or usage. Inactive clauses in Tier2 are moved to Local where the deletion is performed. Periodically, all the clauses in Local are sorted and deleted based on their activity.

6.2 Online Clause Deletion

Our online clause deletion scheme that we propose to replace the delete half scheme is as follows. The TEMP clauses (normally in Local) are maintained in a circular list L with an index variable i that traverses the list in one direction. The index identifies the current "deletion candidate" L_i. We assume a clause quality measure Q, and some threshold quality value q and we want to make sure clauses with quality higher than this threshold will not be deleted while their quality remains high. When a new learned clause C needs to be stored in L, we select a "low quality" clause in the list to be replaced with C by sequential search. As long as $Q(L_i) \geq q$, we increment i to look for the next candidate. This means "saving" clause L_i for one more "round". The first time $Q(L_i) < q$, we replace L_i with C and delete the "old" clause, L_i. The clause quality measure threshold must be chosen so that there are always sufficiently many low-quality clauses in the list (which we will continue to call Local) otherwise there will be no more room for new learned clauses and in a sense the list L becomes a database for PERM clauses. There are algorithmic methods to ensure this (for example, using a feedback control mechanism) but in our experiments, it was not hard to obtain good practical performance without them.

We call our scheme "online", in analogy to online algorithms, because each time a conflict clause is derived we make a decision about which previously learned clause to replace it with, having no information about future clauses. It is not an online algorithm in the usual sense of there being a dichotomy between making decisions as data is read and making decisions after all data is obtained. In that sense, all clause deletion schemes sit somewhere in between.

6.2.1 Relating Delete-Half and Online Deletion

Consider a Delete-Half scheme from Local with a sort-and-reduce phase every k conflicts. Roughly speaking (ignoring some details for simplicity) each TEMP clause is inspected every k conflicts, deleted if its quality is below the median of the current clauses in Local. If we instantiate the size of the list L in our online scheme with $S = 2k$, and keep q sufficiently close to the median, we expect each clause to be inspected every k conflicts and deleted if its quality is below the median of the current TEMP clauses. In this sense, the two schemes can be made quite close: we trade off sorting for dynamically estimating the median. In doing so, we get a clause database of uniform size, rather than one that significantly grows and shrinks.

6.3 Age-Based Deletion

A trivially implemented version of our scheme assumes $Q(C) < q$ for every clause C. This results in a pure age-based scheme: Each new learned clause replaces the oldest learned clause in L. This very low-cost scheme works surprisingly well. Figure 6.2 shows a "cactus-

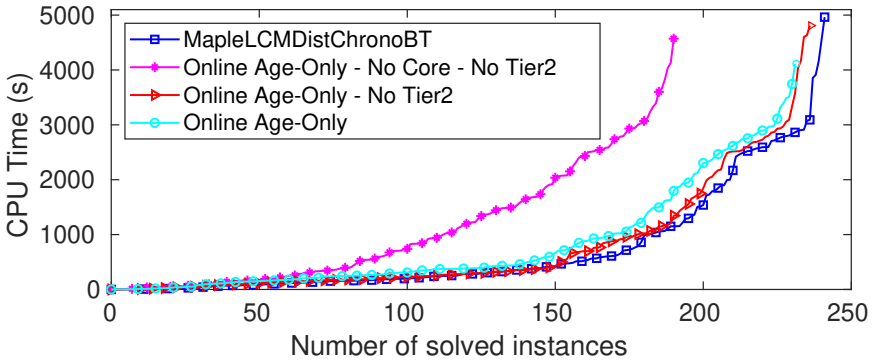


Figure 6.2: Simple Online Deletion Performance.

plot" comparison of default MapleLCMDistChronoBT with 3 variants using online deletion. In this plot, we also illustrate the importance of having a Core DB to store PERM clauses again. The size limit of Local is set to 80,000 clauses in all solvers using online deletion reported here. This results in having at least 80,000 clauses at a time stored by the solver to work with (after the first time number of clauses in Local reaches this value).
Here is a brief description of the 3 modified solvers with online deletion reported in figure 6.2:

Online Age-Only - No Core, No Tier2 In this version, we store all clauses in Local and perform online deletion there. This has no PERM set at all, just pure age-based deletion of all learned clauses.

Online Age-Only - No Tier2 This version have a PERM set based on LBD and Size of clauses. This keeps clauses with LBD ≤ 3 or Size ≤ 4 permanently in Core, and stores the rest of the clauses in Local where uses a pure online age-based deletion.

Online Age-Only In this version we have Core and Tier2 just as in MapleLCMDistChronoBT, but use age-based online deletion from Local with the maximum size of 80,000 (as in the previous two modifications). If a clause is moved from Tier2 to Local, it replaces the oldest clause in Local similar to adding a new learned clause to Local.

Figure 6.2 confirms our previous observation that having a set of PERM clauses (Core) is important to the performance of MapleLCMDistChronoBT. It also shows that in the presence of Core a simple pure age-based deletion scheme for Local gives quite good performance.

6.4 Clause Usage

MiniSAT and many of its successors, including MapleLCMDistChronoBT, use clause "activity" scores in their clause deletion schemes [39, 69, 90]. Every time a clause is used in

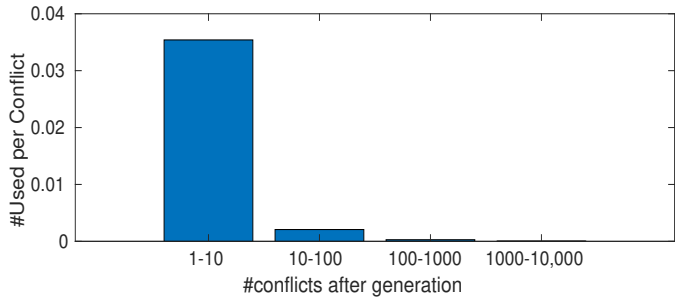


Figure 6.3: Rate of use of clauses in Local at different ages.

conflict analysis, its activity is "bumped", meaning its activity score is increased by a reward value. The reward is initialized to 1 and divided by 0.999 (the decay factor) at each conflict, to simulate decay of activities. This way at each time, the recent usages have more weight on activities than older ones. To prevent activity overflow, when the activity of any clause reaches 1e20, all activity values and the reward value are divided by 1e-20 [39, 23].

This scheme, with many variations, has been widely used, but it also has inconvenient aspects as mentioned before. We anticipated that, in the presence of a separate DB for storing PERM clauses, much simpler usage measures might be effective. We start by an experiment to report the usefullness of learned clause in conflict analysis over time. We make two observations regarding usage and online deletion:

1. We show that Age is highly correlated with usage rate, and can account for a large fraction of clauses that would normally be saved based on clause activities. This is illustrated by Figure 6.3, which shows the average usage rates of clauses that have been in Local for at least 10K conflicts, at different ages. Each bar shows the average number of times a clause is used in conflict analysis after generation divided by the number of conflicts in that bar (y axis). This figure shows that the usage rate, which is an indicator of probability of a clause being used in conflict analysis, is very high in the first 10 conflicts right after generation. As the age of a clause increases, the usage rate of most clauses drops very quickly. This figure shows the usage rate in TEMP clauses but a similar pattern exist in the PERM clauses of MapleLCMDistChronoBT. The only difference is that the usage rate in each bar of PERM clauses is almost twice the values in figure 6.3.

2. In online deletion with Local of size S, if the probability of saving a clause is at most 0.5 (see Figure 6.5), then every learned clause is kept for at least $S/2$ conflicts, giving it substantial time to be used in comparison with Delete-Half schemes. This gives us the flexibility of using a simple quality measure in online deletion.

Recent Usage: We define a few variants of Recent Usage measure for online deletion similar to the Usage definition in chapter 4. We added Recent so that similar to Activity measure, the recent usages in conflict analysis have more influence on the quality of a clause.

Here we report two clause quality measures to be used in online deletion scheme that we have considered. Both are extremely simple to implement. We call them Recent Usage (RU) and Recent Usage Decayed (RUD) and we used them along with online deletion in MapleLCMDistChronoBT to evaluate performance. We continue this section by their descriptions and reports of three experiments that may shed light on the performance of the Recent Usage (RU) measures.

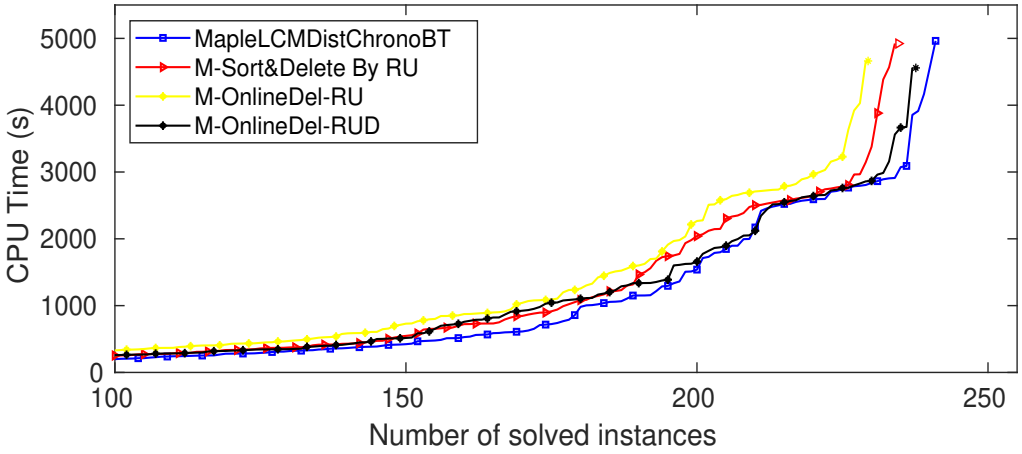


Figure 6.4: Online Deletion with Recent Usage

M-OnlineDel-RU In this version, the measure Q of quality (or activity) is called RU and is just the number of times the clause was used in conflict analysis during the last "round". That is, we update RU count every time the clause is used and reset the count to zero if the clause becomes a candidate for deletion but is saved. We denote this measure RU, for Recently-Used. If the threshold value is q (denoted RU = q), a clause will be saved if it was used q or more times in the last round.

M-OnlineDel-RUD This is similar to M-OnlineDel-RU, but instead of resetting RU to 0 when a clause is considered for deletion and saved, we decay its RU value by dividing it by a constant. We call this measure RUD, for Recently-Used-Decayed.

Figure 6.4 shows the performance of M-OnlineDel-RU with threshold RU=2 and M-OnlineDel-RUD with RU=2 and Decay constant 4. Both versions perform quite well, the decay version being almost as good as MapleLCMDistChronoBT. This suggests that online deletion using simple measures might compete effectively with Delete-Half using traditional activities.

M-Sort&Delete By RU To understand the effectiveness of RU versus traditional activities, we created a solver M-Sort&Delete By RU that is identical to MapleLCMDistChronoBT but does sorting and deletion from Local based on RU instead of activity. The RU value of all clauses in reset to zero after each clause deletion. Figure 6.4 shows the performance is slightly inferior to MapleLCMDistChronoBT on our benchmark, lying between the performance of the two versions with online deletion. This suggests that we pay no penalty for using online deletion instead of the Delete-Half scheme, and confirms that in the presence of a PERM set, a simple usage measure can be almost as useful as traditional clause activities.

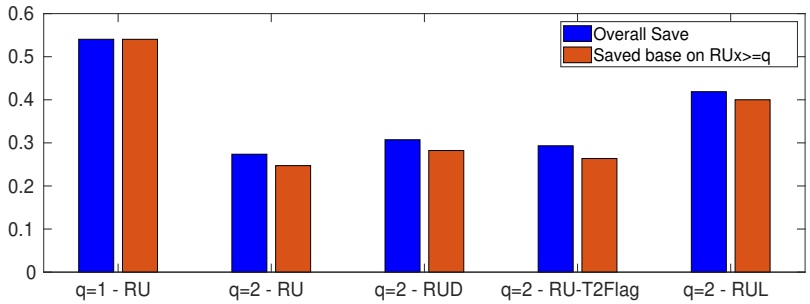


Figure 6.5: Fraction of saved clauses in different online deletion schemes

6.4.1 Fraction Saved by RU

Here we examine the fraction of clauses in Local that become candidates for deletion but are saved based on the RU measure. Figure 6.5 shows this value for several variations. In each pair of bars, the right bar (orange) shows the fraction of clauses with RU $\geq q$; the left bar (blue) shows the fraction of clauses saved based on either RU or because of being "locked" [39]. Locked clauses are the ones that are a reason to the current assignments of the solver and can not be deleted until a backjump or restart is activated to un-assign them.

With $q = 1$, the probability of deletion is almost $1/2$, and the performance of the solver is poor. In contrast, with $q = 2$ using either RU or RUD in online deletion, about three quarters of clauses are deleted, and the performance is quite good as shown in Figure 6.4. We explain other bars shown in this plot later. In the remainder of the experiments in this chapter, all solvers using online deletion with RU measure have q set to 2.

6.4.2 Clauses saved by RU and Activity

We designed an experiment to help us better understand the difference of using RU versus activity in the actual clauses that would be deleted from Local. We examined the clauses in Local just before the 10^{th} clause deletion in MapleLCMDistChronoBT, and measured their RU and activity values to see what fraction of clauses would be saved by our RU-based

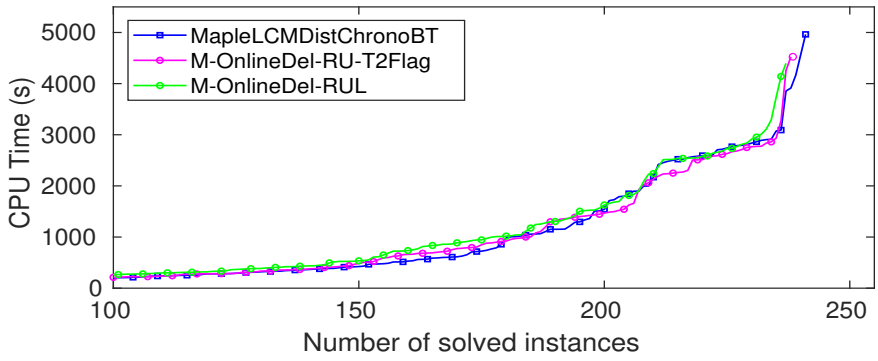


Figure 6.6: Online Deletion With Usage and LBD

schemes. We examined the result of this experiment in 10 different families of the formulas in SAT competition. Table 6.1 shows the results for one formula from each of 10 families. First column determines the formula we are looking at. The second column is the number of clauses in Local just before deletion. Other columns show the number of clauses that would be saved due to RU $\geq q$, and the fraction (in percent) of these clauses that have high enough activity to be saved by Delete-Half. On average this fraction is between 87 and 97 percent, suggesting that simple RU counters can account for a significant fraction of decisions for clause deletion based on activities.

6.5 Clause LBD and Tier2

LBD is used in MapleLCMDistChronoBT for initial placement of a learned clause, and to move clauses between stores if the LBD changes. Here we report two simple modifications to MapleLCMDistChronoBT to take into account these changes in the solver with online deletion and no Tier2. Figure 6.6 show the resulting performance.

Table 6.1: Commonality among High-Activity Clauses and Recently-Used Clauses.

Formula	Local Size	RU ≥ 1 (%)		RU ≥ 2 (%)		RUD ≥ 2 (%)		RUL ≥ 2 (%)	
201	28470	12150	(100)	2162	(100)	2265	(97)	3591	(100)
CNP-5-20	28699	13177	(98)	4122	(100)	4923	(87)	2915	(99)
Karatsuba	25251	11091	(86)	1730	(90)	2111	(80)	5571	(89)
T62.2.0	7097	2474	(100)	521	(100)	618	(86)	1574	(100)
ae_rphp	30535	12586	(87)	6575	(94)	8004	(78)	7278	(94)
apn-sbox6	29422	15459	(87)	6423	(92)	7129	(85)	6282	(90)
cms-scheel	21828	8602	(100)	2454	(100)	2698	(92)	3752	(100)
courses	13869	3241	(100)	854	(100)	1092	(83)	1610	(100)
cz-alt-3-7	26577	11276	(99)	2346	(100)	2639	(93)	7247	(99)
dist9.c	26274	15150	(84)	4182	(92)	4614	(87)	6651	(90)
Average	**23802**	**10521**	**(93)**	**3137**	**(97)**	**3609**	**(87)**	**3395**	**(96)**

M-OnlineDel-RU-T2Flag Here we replace Tier2 with a rough simulation, by adding a "Tier 2 flag" to clauses in Local. We set the flag true if MapleLCMDistChronoBT would move it from Local to Tier2 (but keeping it in Local), and false for the reverse direction. Clauses with this flag true are always saved. This is not an accurate Tier2 simulation, because the size of the clause DB does not change appropriately. Nonetheless, the resulting performance is very close to the original solver.

M-OnlineDel-RUL Here we take LBD into account by modifying the usage scoring. Instead of incrementing RU by 1 each time a clause is used, we increment by c/LBD, for a constant c. We call this RUL, for Recent Usage with LBD. The RUL values are re-set to zero when a clause becomes a candidate for deletion and is saved. The magenta curve in Figure 6.6 shows the performance of this solver with $c = 20$. Choosing this value for c results in clauses with $LBD \leq 10$ to be save if at leased used once (similar to Tier2), clauses with $10 < LBD \leq 20$ to be saved if used twice or more, clauses with $20 < LBD \leq 30$ to be saved if used 3 times or more, and so on.

Looking back at figure 6.5, we can see that the fraction of clauses that are saved from deletion with the M-OnlineDel-RU-T2Flag scheme is similar to simple M-OnlineDel-RU scheme and about 30 percent. This amount increases to almost 40 percent in the M-OnlineDel-RUL scheme.

6.6 Computation Time

Given that online deletion does not need sorting clauses, it is reasonable to expect it to take less time than Delete-Half scheme which is a sorting-based deletion. Here we will illustrate the difference using profile data for a SAT formula from the 2018 competition benchmarks. The formula was chosen from the "Grand Tour Puzzle" family [32] as it takes about the same time (100 seconds) to be solved with both solvers and is easier to meaningfully compare the clause deletion time for. Other formulas showed a similar pattern which is expected as time spent on clause deletion is mostly effected by size of clause database and not structure of the formulas.

We aim to compare the time spent on clause deletion in both Delete-Half and Online Deletion schemes which is shown in figure 6.7. The Delete-Half scheme is the one implemented in default MapleLCMDistChronoBT so we solve the problem with this solver and looked into the deletion from Local. We also solved the problem with M-OnlineDel-RU solver which uses online deletion for comparison.

Figure 6.7 shows the fraction of time spent on each part of the clause deletion scheme in both solvers. The "Garbage Collector" function is the actual removal of clauses and freeing the memory allocated to them. In both solvers it happens every 15,000 conflicts. The "Remove Clause" function is responsible for detaching the deleted clauses from Local.

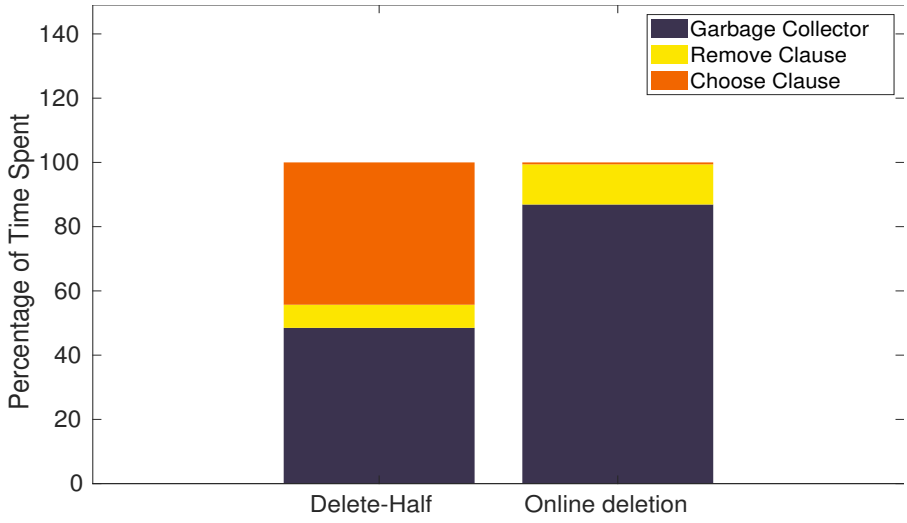


Figure 6.7: Fraction of Time Spent in Various Parts of Clause Deletion Methods

Finally, the "Choose Clause" function is the function that finds clauses for deletion and marks them as deleted. The difference in deletion scheme comes from this function. As illustrated in figure 6.7, more than 40% of the deletion time of Delete-Half scheme is spent on the Choose Clause function whereas this number is only 0.05% in online deletion. We would like to point out the garbage collector function in both solvers is very similar in terms of time consumption and takes less than 1% of the total solving time.

6.7 SAT vs. UNSAT formulas

In the final section, we briefly show the effect of online deletion on satisfibale and unsatisfibale formulas. Table 6.2 shows the total number of formulas solved by the main solvers with online deletion introduced in this section distinguishing between satisfiable (SAT) and unsatisfiable (UNSAT) formulas. The first line in table shows that MapleLCMDistChronoBT solved a total of 241 formulas from the 2018 competition benchmarks from which, 138 were satisfiable and 103 were unsatisfiable. Our simplest solver with online deletion, M-OnlineDel-RU, solved 132 formulas from satisfiable and 98 from the unsatisfiable instance which is proportionally very similar to MapleLCMDistChronoBT if we consider the size of each set. The next three solver in the table suggest that by improving the clause quality criteria in the online deletion, we could improve the performance on satisfiable formulas and even outperforming MapleLCMDistChronoBT but this improvement is not observed on unsatisfiable formulas.

Table 6.2: Performance on Satisfiable *vs* Unsatisfiable Formulas.

Solver	# Solved	SAT	UNSAT
MapleLCMDistChronoBT	**241**	138	**103**
M-OnlineDel-RU	230	132	98
M-OnlineDel-RUL	237	140	97
M-OnlineDel-RUD	238	**141**	97
M-OnlineDel-RU-T2Flag	238	139	99

6.8 Summary

In this chapter, we started by pointing out to some of the complexities of the most popular clauses database management strategies and we proposed a new, simple clause deletion scheme called online deletion. This helps the solver to maintain a set of TEMP clauses of fixed size and every time a new clause is learned, it will replace a current clause with lower expected quality.
We follow by reporting the performance of different instantiations of the online deletion using clause age, LBD and very simple measures of usage as clause quality. An implementation of the online scheme in MapleLCMDistChronoBT, the winning solver from the main track of the 2018 SAT solver competition, has performance almost as good as the original solver.

We show that online deletion requires less computation time than the Delete-Half scheme. However, the fraction of run time consumed by deletion in MapleLCMDistChronoBT is small, so this is not a major performance factor but can be helpful in understanding the effect of different parameters of deletion scheme on performance.
We show that usage rate in recently learned clauses is high and so the online deletion schemes we use here consider age or age modified by a fixed quality threshold to take that into account. A dynamic threshold may be more desirable, in which case we may use a feedback control scheme to ensure the threshold is such that the fraction of saved clauses is suitable. We replace the activity with a simple recent usage measure and show that it accounts for most of the clauses saved in the deletion scheme of MapleLCMDistChronoBT.

Finally, we show a comparison of the solvers on satisfiable vs. unsatisfiable formulas and show that some versions of online deletion work better on satisfiable instances but not unsatisfiable ones.

Chapter 7

Conclusion

In this dissertation, we contributed in two main directions. The first is introducing a new measure based on the structure of SAT formulas, showing it is meaninful in various ways, and utilizing it to modify CDCL solver heuristics to improve solver performance. The second is reviewing current clause database management schemes to understand different components of them better and proposing methods to replace and improve the current deletion schemes.

- CDCL solver input formulas are in CNF, and generating the primal graph of those formulas is a common way of studying the structure of them. We introduced a new structural measure called centrality based on the betweenness centrality of variables in primal graph of the input CNF formulas. We demonstrated that the primal graphs of some industrial formulas have nice "coarse structure" that can be exploited fairly easily. We have extended previous observations that the VSIDS decision heuristic in CDCL solvers shows a strong preference for particular families of variables and showed this applies to variables with high centralities as well. Wondering whether this was good or bad led us to experiment with preferential bumping which is a new scheme proposed by us to change the preference of VSIDS decisions for a special set of variables. We have shown that the performance of Glucose can be meaningfully improved by preferential bumping schemes. We continued the experiments modifying more complicated solvers from the MapleSAT solver family to see if our findings can be used in other solvers. We introduced three centrality-based modifications to standard CDCL decision heuristics, VSIDS and LRB, and implemented these in Maple LCM Dist, the first-place solver from the main track of the 2017 SAT Solver Competition. All three changes improved the performance on the industrial formulas. The centrality based modifications seem to be interesting as they seem to work for different formulas and even result in solving formulas that no other solver in competition could solve within the time limit.

- Considering the effectiveness of using centrality in the decision heuristics of CDCL solvers, we introduced centrality-based deletion schemes to utilize centrality in clause

database management heuristics. This deletion scheme is based on clause centrality, a new measure of clause quality introduced here. We presented evidence that clause centrality is an interesting quality measure and showed that the centrality of clauses is correlated with their usefulness in conflict analysis. We implemented clause deletion schemes based on centrality in two different solvers that were the winners of 2017 and 2018 SAT competitions and showed their performance was improved. Future directions could include more in-depth study of the roles of variable and clause centrality in solver execution, and development of a centrality-based restart strategy. While we feel we have made some progress in understanding the role of structure-based measures in CDCL solvers, and have found some promising techniques for improving solver performance, we cannot claim to have shed light on the question of what makes CDCL solvers preference toward these variables.

- We studied the clause database management heuristic in terms of current definitions of permanent and temporary clauses. We started by illustrating the important of having a set of permanent clauses that are mainly small clauses. We showed that even though the number of permanent clauses can grow to be very large in some formulas, which is usually not desired, our attempts to reduce the size of this set proved to be harmful to performance. We continued by comparing various definition of permanent clauses based on size and LBD and showed in both measures this set should not be defined too restrictive or too broad for best performance results. We showed storing high centrality clauses permanently is beneficial to solvers. We also showed that there are additional small clauses other than the UIP scheme that can be learned efficiently by the solver with low cost. These can be beneficial to solver when added to the temporary set of clauses, but not to the permanent set. We hope our work sheds some light on the importance of permanent clauses in clause database management heuristics and motivates future work in this area.

- We introduced a new, simple online clause deletion scheme, and reported the performance of instantiations of the scheme using clause age, LBD and very simple measures of usage in conflict analysis. An implementation of the online scheme in MapleLCMDistChronoBT, the winning solver from the main track of the 2018 SAT Competition, has performance almost as good as the original. Online deletion requires less computation time than the Delete-Half scheme. However, the fraction of run time consumed by deletion in MapleLCMDistChronoBT is small, so this is not a major performance factor. The online deletion schemes in this paper use age or age modified by a fixed quality threshold. A dynamic threshold may be more desirable, in which case we may use a feedback control scheme to ensure the threshold is such that the fraction of saved clauses is suitable. Future work directions could investigate more refined versions of online deletion, in particular with regard to clause quality measures

and clause database size. We also showed that our modified solvers are biased toward satisfiable instances, and some work to shift this bias is desired.

Bibliography

[1] The international SAT competitions. `http://www.satcompetition.org/`.

[2] PeneLoPe, a parallel SAT solver. `http://www.cril.univ-artois.fr/~hoessen/penelope.html/`.

[3] Michael Alekhnovich and Alexander A. Razborov. Satisfiability, branch-width and Tseitin tautologies. In *43rd Symposium on Foundations of Computer Science (FOCS)*, pages 593–603. IEEE, 2002.

[4] Carlos Ansótegui, Maria Luisa Bonet, Jesús Giráldez-Cru, and Jordi Levy. Community structure in industrial SAT instances. *arXive preprint arXiv:1606.03329*, 2016.

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

[6] 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 (SAT)*, pages 238–254. Springer, 2015.

[7] Albert Atserias, Johannes Klaus Fichte, and Marc Thurley. Clause-learning algorithms with many restarts and bounded-width resolution. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 114–127. Springer, 2009.

[8] Gilles Audemard and Laurent Simon. Glucose: a solver that predicts learnt clauses quality. *SAT Competition*, pages 7–8, 2009.

[9] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *International Joint Conference on Artifical Intelligence (IJCAI)*, pages 399–404. Morgan Kaufmann, 2009.

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

[11] Gilles Audemard and Laurent Simon. Lazy clause exchange policy for parallel SAT solvers. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 197–205. Springer, 2014.

[12] Tomas Balyo, Nils Froleyks, Marijn J.H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Proceedings of SAT competition 2020: Solver and benchmark descriptions. Technical report, University of Helsinki, 2020.

[13] Tomas Balyo, Marijn J.H. Heule, and Matti Järvisalo. Proceedings of SAT competition 2016: Solver and benchmark descriptions. Technical report, University of Helsinki, 2016.

[14] Tomas Balyo, Marijn J.H. Heule, and Matti Järvisalo. SAT competition 2016: Recent developments. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[15] Luís Baptista and Joao Marques-Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In *International conference on principles and practice of constraint programming (CP)*, pages 489–494. Springer, 2000.

[16] Eli Ben-Sasson and Avi Wigderson. Short proofs are narrow — resolution made simple. In *Thirty-first annual ACM symposium on Theory of computing*, pages 517–526. ACM, 1999.

[17] Armin Biere. Adaptive restart strategies for conflict driven SAT solvers. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 28–33. Springer, 2008.

[18] Armin Biere. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4(2-4):75–97, 2008.

[19] Armin Biere. PrecoSAT solver description for SAT competition 2009. *SAT Competitive Event Booklet*, 2009.

[20] Armin Biere. Lingeling and friends at the SAT competition 2011. *FMV Report Series*, 11(1), 2011.

[21] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 193–207. Springer, 1999.

[22] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In *Proceedings of SAT competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1, pages 51–53. University of Helsinki, 2020.

[23] Armin Biere and Andreas Fröhlich. Evaluating CDCL variable scoring schemes. In *International Conference of Theory and Applications of Satisfiability Testing (SAT)*, pages 405–422. Springer, 2015.

[24] Armin Biere, Marijn J.H. Heule, and Hans van Maaren. *Handbook of Satisfiability*, volume 185. IOS press, 2009.

[25] James R. Bitner and Edward M. Reingold. Backtrack programming techniques. *Communications of the ACM*, 18(11):651–656, 1975.

[26] Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *The Journal of statistical mechanics: theory and experiment*, 2008(10):10008, 2008.

[27] Phillip Bonacich. Factoring and weighting approaches to status scores and clique identification. *The Journal of Mathematical Sociology*, 2(1):113–120, 1972.

[28] Ulrik Brandes. A faster algorithm for betweenness centrality. *The Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[29] Robert G. Brown. Exponential smoothing for predicting demand. In *Operations Research*, volume 5, pages 145–145, 1957.

[30] Michael Buro and H Kleine Büning. *Report on a SAT competition*. Fachbereich Math. Informatik, University of Gesamthochschule, 1992.

[31] Cedar, A Compute Canada Cluster. `https://docs.computecanada.ca/wiki/Cedar`.

[32] Md Solimul Chowdhury, Martin Müller, and Jia-Huai You. GrandTourobs puzzle as a SAT benchmark. *Proceedings of SAT Competition*, pages 59–60, 2018.

[33] Compute Canada: Advanced Research Computing (ARC) Systems. `https://www.computecanada.ca/`.

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

[35] James M. Crawford and Larry D. Auton. Experimental results on the crossover point in satisfiability problems. In *AAAI*, volume 93, pages 21–27. Citeseer, 1993.

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

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

[38] Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. Towards a better understanding of the functionality of a conflict-driven SAT solver. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 287–293. Springer, 2007.

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

[40] Herbert B. Enderton. *A mathematical introduction to logic*. Elsevier, 2001.

[41] Santo Fortunato. Community detection in graphs. *Physics reports*, 486(3-5):75–174, 2010.

[42] Jon W. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, University of Pennsylvania, 1995.

[43] Linton C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, 1977.

[44] Zhaohui Fu, Yogesh Mahajan, and Sharad Malik. New features of the SAT04 versions of zChaff. *SAT Competition*, 2004.

[45] Michael R. Garey and David S. Johnson. *Computers and intractability*, volume 174. freeman San Francisco, 1979.

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

[47] Carla P. Gomes and Bart Selman. Problem structure in the presence of perturbations. In *National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, (AAAI)*, pages 221–226. AAAI, 1997.

[48] Carla P. Gomes, Bart Selman, and Nuno Crato. Heavy-tailed distributions in combinatorial search. In *International Conference on Principles and Practice of Constraint Programming (CP)*, pages 121–135. Springer, 1997.

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

[50] Carla P. Gomes, Bart Selman, Ken McAloon, and Carol Tretkoff. Randomization in backtrack search: Exploiting heavy-tailed profiles for solving hard scheduling problems. In *Artificial Intelligence Planning Systems (AIPS)*, pages 208–213, 1998.

[51] Shai Haim and Marijn J.H. Heule. Towards ultra rapid restarts. *arXiv preprint arXiv:1402.4413*, 2014.

[52] Marijn J.H. Heule, Matti Järvisalo, and Martin Suda. Proceedings of SAT race 2019: Solver and benchmark descriptions. Technical report, University of Helsinki, 2019.

[53] Marijn J.H. Heule, Matti Järvisalo, and Martin Suda. SAT competition 2018. *Journal on Satisfiability, Boolean Modeling and Computation*, 11(1):133–154, 2019.

[54] Jinbo Huang. A case for simple SAT solvers. In *International Conference on Principles and Practice of Constraint Programming (CP)*, pages 839–846. Springer, 2007.

[55] Sima Jamali and David Mitchell. Improving SAT solver performance with structure-based preferential bumping. In *Global Conference of Artificial Intelligence (GCAI)*, pages 175–187. Springer, 2017.

[56] Sima Jamali and David Mitchell. Centrality-based improvements to CDCL heuristics. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 122–131. Springer, 2018.

[57] Sima Jamali and David Mitchell. Maple LCM OnlineDel. *SAT RACE*, page 27, 2019.

[58] Sima Jamali and David Mitchell. Simplifying CDCL clause database reduction. In *International Conference on Theory and Applications of Satisfiability Testing SAT*, pages 183–192. Springer, 2019.

[59] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The international SAT solver competitions. *AI Magazine*, 33(1):89–94, 2012.

[60] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of mathematics and Artificial Intelligence*, 1(1-4):167–187, 1990.

[61] George Katsirelos and Laurent Simon. Eigenvector centrality in industrial SAT instances. In *International Conference on Principles and Practice of Constraint Programming (CP)*, pages 348–356. Springer, 2012.

[62] Henry Kautz and Bart Selman. Planning as satisfiability. In *European Conference on Artificial Intelligence (ECAI)*, volume 92, pages 359–363, 1992.

[63] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the national conference on artificial intelligence*, pages 1194–1201, 1996.

[64] Stepan Kochemazov, Oleg Zaikin, Victor Kondratiev, and Alexander Semenov. MapleLCMDistChronoBT-DL, duplicate learnts heuristic-aided solvers at the SAT race. *Proceedings of SAT Race*, pages 24–24, 2019.

[65] Markus Krötzsch. *Description logic rules*, volume 8. IOS Press, 2010.

[66] Daniel Le Berre and Laurent Simon. Fifty-five solvers in Vancouver: The SAT 2004 competition. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 321–344. Springer, 2004.

[67] Chunxiao Li, Jonathan Chung, Soham Mukherjee, Marc Vinyals, Noah Fleming, Antonina Kolokolova, Alice Mu, and Vijay Ganesh. On the hierarchical community structure of practical boolean formulas. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 359–376. Springer, 2021.

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

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

[70] Jia Hui Liang, Vijay Ganesh, Ed Zulkoski, Atulan Zaman, and Krzysztof Czarnecki. Understanding VSIDS branching heuristics in conflict-driven clause-learning SAT solvers. In *Hardware and Software: Verification and Testing: Haifa Verification Conference*, pages 225–241. Springer, 2015.

[71] Jia Hui Liang, Chanseok Oh, Vijay Ganesh, Krzysztof Czarnecki, and Pascal Poupart. Maple-COMSPS, MapleCOMSPS LRB, MapleCOMSPS CHB. *SAT Competition*, page 52, 2016.

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

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

[74] Mao Luo, Chu-Min Li, Fan Xiao, Felip Manyà, and Zhipeng Lü. An effective learnt clause minimization approach for CDCL SAT solvers. In *26th International Joint Conference on Artificial Intelligence (IJCAI 2017), Melbourne, Australia, August 19–25, 2017*, pages 703–711. ijcai.org, 2017.

[75] Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik. Zchaff2004: An efficient SAT solver. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 360–375. Springer, 2004.

[76] Joao Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Portuguese Conference on Artificial Intelligence*, pages 62–74. Springer, 1999.

[77] Joao Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In *Handbook of satisfiability*, pages 131–153. IOS Press, 2009.

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

[79] Ruben Martins, Vasco Manquinho, and Inês Lynce. Community-based partitioning for MaxSAT solving. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 182–191. Springer, 2013.

[80] Robert Mateescu. Treewidth in industrial SAT benchmarks. Technical Report MSR-TR-2011-22, Microsoft, February 2011.

[81] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of Annual Design Automation Conference (DAC)*, pages 530–535. ACM, 2001.

[82] Alexander Nadel, Moran Gordon, Amit Palti, and Ziyad Hanna. Eureka-2006 SAT solver. *Solver description - SAT Race*, 2006.

[83] Alexander Nadel and Ryvchin Vadim. Chronological backtracking. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 111–121. Springer, 2018.

[84] NetworkX, Software for complex networks. `https://networkx.github.io/`.

[85] Miguel Neves, Ruben Martins, Mikoláš Janota, Inês Lynce, and Vasco Manquinho. Exploiting resolution-based representations for MaxSAT solving. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 272–286. Springer, 2015.

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

[87] 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 (SAT)*, pages 252–268. Springer, 2014.

[88] 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 (SAT)*, pages 62–70. Springer, 2015.

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

[90] Chanseok Oh. *Improving SAT solvers by exploiting empirical characteristics of CDCL*. PhD thesis, New York University, 2016.

[91] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *International conference on theory and applications of satisfiability testing (SAT)*, pages 294–299. Springer, 2007.

[92] Knot Pipatsrisawat and Adnan Darwiche. Rsat 2.0: SAT solver description. Technical report, Citeseer, 2007.

[93] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.

[94] Lawrence Ryan. *Efficient algorithms for clause-learning SAT solvers*. PhD thesis, Simon Fraser University, 2004.

[95] Vadim Ryvchin and Ofer Strichman. Local restarts in SAT. *Constraint Programming Letters (CPL)*, 4:3–13, 2008.

[96] SAT Competition 2017. `https://baldur.iti.kit.edu/sat-competition-2017/`, July 2017.

[97] Thomas J Schaefer. The complexity of satisfiability problems. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 216–226, 1978.

[98] Joao Marques Silva and Karem A. Sakallah. Grasp—a new search algorithm for satisfiability. 1996.

[99] Joao Marques Silva and Karem A. Sakallah. Grasp—a new search algorithm for satisfiability. In *The Best of ICCAD*, pages 73–89. Springer, 2003.

[100] Tomohiro Sonobe, Shuya Kondoh, and Mary Inaba. Community branching for parallel portfolio SAT solvers. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 188–196. Springer, 2014.

[101] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 244–257. Springer, 2009.

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

[103] Niklas Sörensson and Niklas Een. Minisat v1. 13 - a SAT solver with conflict-clause minimization. Poster at the International Conference on Theory and Applications of Satisfiability Testing (SAT), 2005.

[104] Niklas Sörensson and Niklas Eén. Minisat 2.1 and minisat++ 1.0. *SAT Race*, page 31, 2009.

[105] Paul Stephan, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(9):1167–1176, 1996.

[106] Richard S Sutton and Andrew G Barto. Reinforcement learning: An introduction. 2011.

[107] Grigori S Tseitin. On the complexity of derivation in propositional calculus. In *Automation of reasoning*, pages 466–483. Springer, 1983.

[108] Vincent Vallade, Ludovic Le Frioux, Souheib Baarir, Julien Sopena, Vijay Ganesh, and Fabrice Kordon. Community and LBD-based clause sharing policy for parallel SAT solving. In *International Conference on Theory and Applications of Satisfiability Testing(SAT)*, pages 11–27. Springer, 2020.

[109] Toby Walsh. Search in a small world. *IJCAI*, 99:1172–1177, 1999.

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

[111] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008.

[112] Ramin Zabih and David A. McAllester. A rearrangement search strategy for determining propositional satisfiability. In *AAAI*, volume 88, pages 155–160, 1988.

[113] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 279–285. IEEE, 2001.

[114] Xindi Zhang and Shaowei Cai. Relaxed backtracking with rephasing. *SAT competition 2020*, page 15, 2020.